



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

A Component Architecture for High-Performance Scientific Computing

D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, S. Zhou

December 16, 2004

International Journal of High-Performance Computing
Applications

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

A Component Architecture for High-Performance Scientific Computing

David E. Bernholdt¹ Benjamin A. Allan² Robert Armstrong² Felipe Bertrand³ Kenneth
Chiu³ Tamara L. Dahlgren⁴ Kostadin Damevski⁵ Wael R. Elwasif¹ Thomas
G. W. Epperly⁴ Madhusudhan Govindaraju⁶ Daniel S. Katz⁷ James A. Kohl¹ Manoj
Krishnan⁸ Gary Kumfert⁴ J. Walter Larson⁹ Sophia Lefantzi¹⁰ Michael J. Lewis⁶ Allen D.
Malony¹¹ Lois C. McInnes⁹ Jarek Nieplocha⁸ Boyana Norris⁹ Steven G. Parker⁵ Jaideep
Ray¹² Sameer Shende¹¹ Theresa L. Windus¹³ and Shujia Zhou¹⁴

Submitted to *Intl. J. High Performance Computing Applications*, ACTS Collection Special Issue

¹Computer Science and Mathematics Division, Oak Ridge National Laboratory, P. O. Box 2008, Oak Ridge, TN 37831

²Scalable Computing R & D, MS 9915, PO Box 969, Sandia National Laboratories, Livermore, CA 94551-0969

³215 Lindley Hall, Indiana University, 47405

⁴Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P.O. Box 808, L-365, Livermore, CA 94551

⁵Scientific Computing and Imaging Institute, University of Utah, 50 S. Central Campus Dr., Room 3490, Salt Lake City, UT 84112

⁶Department of Computer Science, State University of New York (SUNY) at Binghamton, Binghamton, NY 13902

⁷Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91109

⁸Computational Sciences and Mathematics, Pacific Northwest National Laboratory, Richland, WA 99352

⁹Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Ave., Argonne, IL 60439-4844

¹⁰Reacting Flow Research, MS 9051, PO Box 969, Sandia National Laboratories, Livermore, CA 94551-0969

¹¹Department of Computer and Information Science, University of Oregon, Eugene, OR 97403

¹²Advanced Software R & D, MS 9051, PO Box 969, Sandia National Laboratories, Livermore, CA 94551-0969

¹³Pacific Northwest National Laboratory, Environmental Molecular Sciences Laboratory, P.O. Box 999, MS-IN: K8-91, Richland, WA 99352

¹⁴Northrop Grumman IT/TASC, 4801 Stonecroft Blvd, Chantilly, VA 20151

Corresponding Author:

David E. Bernholdt

Computer Science and Mathematics Division

Oak Ridge National Laboratory

P. O. Box 2008, MS 6016

Oak Ridge, TN 37831-6016 USA

Phone: +1 865 574 3147

Fax: +1 865 576 5491

Email: *bernholdtde@ornl.gov*

Abstract

The Common Component Architecture (CCA) provides a means for software developers to manage the complexity of large-scale scientific simulations and to move toward a *plug-and-play* environment for high-performance computing. In the scientific computing context, component models also promote collaboration using independently developed software, thereby allowing particular individuals or groups to focus on the aspects of greatest interest to them. The CCA supports parallel and distributed computing as well as local high-performance connections between components in a language-independent manner. The design places minimal requirements on components and thus facilitates the integration of existing code into the CCA environment. The CCA model imposes minimal overhead to minimize the impact on application performance. The focus on high performance distinguishes the CCA from most other component models. The CCA is being applied within an increasing range of disciplines, including combustion research, global climate simulation, and computational chemistry.

1 Introduction

Historically, the principle concerns of software developers for high-performance scientific computing have centered on increasing the scope and fidelity of their simulations, and then increasing the performance and efficiency to address the exceedingly long execution times that can accompany these goals. Initial successes with computational simulations have led to the desire for solutions to larger, more sophisticated problems and the improvement of models to reflect greater levels of detail and accuracy. Efforts to address these new demands necessarily have included improvements to scientific methodology, algorithms, and programming models, and virtually always each advance has been accompanied by increases in the complexity of the underlying software. At the same time, the computer industry has continued to create ever larger and more complex hardware in an attempt to satisfy the increasing demand for simulation capabilities. These architectures tend to exacerbate the complexity of software running on these systems, as in nearly all cases, the increased complexity is exposed to the programmer at some level and must be explicitly managed to extract the maximum possible performance. In scientific high-performance computing, relatively little attention has been paid to improving the fundamental software development process and finding ways to manage the ballooning complexity of the software and operating environment.

Simultaneously, in other domains of software development, complexity rather than runtime performance has been a primary concern. For example, in the business area the push has been less for increasing the size of “the problem” than for interconnecting and integrating an ever-increasing number of applications that share and manipulate business-related information, such as word processors, spreadsheets, databases, and web servers. More recently, with the fast pace of the internet boom, the flood of new software technology used in business and commercial applications has increased both the degree of interoperability desired and the number of applications and tools to be integrated. This situation has led to extreme software complexity, which at several levels is not unlike the complexity now being seen in high-performance scientific computing. For example, scientific codes regularly attempt the integration of multiple numerical libraries and/or programming models into a single application. Recently, efforts are increasing to couple multiple stand-alone simulations together into multi-physics and multi-scale applications for models with better overall physical fidelity.

One of the approaches that the business/internet software community has found invaluable in helping to address their complexity conundrum is the concept of component-based software engineering (CBSE). The basic tenet of

CBSE is the encapsulation of useful units of software functionality into *components*. Components are defined by the interfaces that they present to the outside world (i.e., to other components), while their internal implementations remain opaque. Components interact *only* through these well-defined interfaces, and based on these interfaces can be composed into full applications. Using this methodology enables use of the “plug-and-play” software approach for creating complex applications. The smaller, task-specific units of software are more easily managed and understood than a complete application software structure. Logically, many components can provide functionality that is generally useful in a variety of different applications. In such cases, suitably well-designed components might be *reusable* across multiple applications with little or no modification. It is also possible that a number of different components can export the same interface and provide the same essential functionality, but via different implementations, thereby allowing these *interoperable* components to be swapped within an application in a plug-and-play fashion.

These ideas have spawned a number of component architectures, some of which have become so widely used as to have reached “commodity” status: Microsoft’s Component Object Model (COM) [Microsoft Corporation, 1999], the Object Management Group’s Common Object Request Broker Architecture (CORBA) Component Model [Object Management Group, 2002], and Sun’s Enterprise JavaBeans [Sun Microsystems, 2004a].

While having proven quite popular and successful in the areas in which it originated, CBSE has made few inroads into the scientific computing community. Part of the reason is that serious scientific applications tend to be large, evolving, and long-lived codes whose lifetimes often extend over decades. In addition to a natural inertia that slows the adoption of new software engineering paradigms by scientific software developers, current commodity component models present a variety of issues ranging from the amount of code that must be changed and added to adapt existing code to the component environment, to performance overheads, to support for languages, data types, and even operating systems widely used in scientific computing.

The Common Component Architecture (CCA) Forum was launched in 1998 as a grass-roots effort to create a component model specifically tailored to the needs of high-performance scientific computing. The group’s goals are both to facilitate scientific application development and to gain a deeper understanding of the requirements for and use of CBSE in this community so that they can feed back into the development of future component models, in the hope that eventually this community too may be adequately served by “commodity” tools. In the intervening years, the Forum has developed a specification for the CCA as well as prototype implementations of many associated tools,

and the CCA is experiencing increasing adoption by applications developers in a number of scientific disciplines. This paper updates our previous overview paper in 1999 [Armstrong et al., 1998] to reflect the evolution of the CCA and our much more detailed understanding of the role of CBSE in scientific computing. We will discuss the special needs of this community, and we will describe in detail the Common Component Architecture and how it addresses these needs, as well as how it relates to other component models. We will provide some information about how scientific applications are using the CCA now, and we will discuss open issues for CBSE development in this area.

2 Components for Scientific Computing

Component-based software engineering is a natural approach for modern scientific computing. The ability to easily reuse interoperable components in multiple applications and the plug-and-play assembly of those applications have significant benefits in terms of productivity in the creation of simulation software, especially when the component “ecosystem” is rich enough that a large portion of the components needed by any given application will be available “off the shelf” from a component repository. The simplicity of plug-and-play composition of applications and the fact that components hide implementation details and provide more manageable units of software development, testing, and distribution all help to deal with the complexity inherent in modern scientific software. Once the overall architecture of a software system and interfaces between elements of it (components) have been defined, individual software developers or small groups can then focus on the creation of the components of particular scientific interest to them, while reusing software developed by others (often experts in their own domains) for other needed components of the system. Componentization also indirectly assists with the performance issues that are so critical in this area by providing the ability to swap components with different implementations that are tailored to the platform of interest. Finally, components are a natural approach to handle the coupling of codes involving different physical phenomena as well as different time scales and length scales, which is becoming increasingly important as a means to improve the fidelity of simulations.

In the scientific software world, CBSE is perhaps most easily understood as an evolution of the widespread practice of using a variety of software libraries as the foundation on which to build applications. Traditional libraries already offer some of the advantages attributed above to components, but a component-based approach extends and magnifies the benefits. While it is possible for users to discover and use library routines that were not meant to be exposed

as part of the library's public interface, component environments can enforce the public interface rigorously. It is possible to have multiple instances (versions) of a component in a component-based application, whereas with plain libraries, this is not generally possible. Finally, in a library-based environment, there are often conflicts of resource management, programming models, or other dependencies, which are more likely as increasing numbers of libraries are involved. With components, these concerns can often be handled by hooking up each "library" component to components providing the resource management or programming model functionality.

The use of domain-specific computational frameworks is another point of contact between current practice in scientific computing and CBSE. Domain-specific frameworks have become increasingly popular as environments in which a variety of applications in a given scientific domain can be constructed. Typically, the "framework" provides a deep computational infrastructure to support calculations in the domain of interest. Applications are then constructed as relatively high-level code utilizing the domain-specific and more general capabilities provided by the framework. Many frameworks support modular construction of applications in a fashion very similar to that provided by component architectures but typically limited to the higher-level parts of the code. However, domain-specific frameworks have their limitations as well. Their domain focus tends to lead to assumptions about the architecture and workflow of the application becoming embodied in the design of the framework, making it much harder to generalize them to other domains. Similarly, since their development is generally driven by a small group of domain experts, it is rare to find interfaces or code that can be easily shared across multiple domain-specific frameworks. Unfortunately, this situation is often true even with important cross-cutting infrastructure, such as linear algebra software, which could in principle be used with common interfaces across many scientific domains. Generic component models, on the other hand, provide all the benefits of domain-specific frameworks, but by casting the computational infrastructure as well as the high-level physics of the applications as components, they also provide easier extension to new areas, easier coupling of applications to create multi-scale and multi-physics simulations, and significantly more opportunities to reuse elements of the software infrastructure.

However, scientific computing also places certain demands on a CBSE environment, some of which are not easily satisfied by most of the commodity component models currently available. As noted above, performance is a paramount issue in modern scientific computing, so that to be acceptable in the community, a component model for scientific computing would have to be able to maintain the performance of traditional applications without imposing

undue overheads. (A widely used rule of thumb is that environments that impose a performance penalty in excess of ten percent will be summarily rejected by high-performance computing (HPC) software developers.) Commodity component models have been designed (primarily or exclusively) for distributed computing and tend to use protocols that assume all method invocations between components must be made over the network, in environments where network latencies are often measured in tens and hundreds of milliseconds. This situation is in stark contrast to HPC scientific computing, where latencies on parallel interconnects are measured in microseconds, and traditional programming practices assume that on a given process in a parallel application, data can be “transferred” between methods by direct reference to the memory location in which it lives. The performance overheads of the commodity component models are often too high for scientific computing.

In scientific computing, it is common to have large codes that evolve over the course of many years or even decades. Therefore, the ease with which “legacy” code bases can be incorporated into a component-based environment, and the cost of doing so, are also important considerations. Many of the commodity component models may require significant restructuring of code and the addition of new code to satisfy the requirements of the model.

Finally, also important to high-performance scientific computing are considerations including support for languages, data types, and computing platforms. The various commodity component models available may not support Fortran or may require extensive use of Java, may not support arrays as first-class data types or complex numbers, and may even be effectively limited to Windows-based operating systems, which are not widely used in scientific HPC.

The Common Component Architecture (CCA) was conceived to remedy the fact that a suitable component environment could not be found to satisfy the special needs of this community. Since its inception, the CCA effort has grown to encompass a community of researchers, several funded projects, an increasing understanding of the role of CBSE in high-performance scientific computing, a maturing specification for the CCA component model, and practical implementations of tools and applications conforming to that specification.

3 The Common Component Architecture

Formally, the Common Component Architecture is a specification of an HPC-friendly component model. This specification also provides a focus for an extensive research and development effort. On the research side, the effort emphasizes understanding how best to utilize and implement component-based software engineering practices in the

high-performance scientific computing arena, and feeding back that information into the broader component software field. In addition to defining the specification, the development effort creates practical reference implementations and helps scientific software developers use them to create CCA-compliant software, and ultimately, a rich marketplace of scientific components, thereby allowing new component-based applications to be built from predominantly off-the-shelf scientific components.

3.1 Philosophy and Objectives

The purpose of the CCA is to facilitate and promote high-performance, high-quality, and more productive scientific software development in a way that is as simple and natural as possible for scientific software developers. The CCA intentionally has much in common with commodity component models but does not hesitate to do things differently where the needs of HPC dictate (see Section 9 for a more detailed comparison). High performance and ease of use are more strongly emphasized in the CCA effort than in commodity component models. However, in principal no barriers exist to providing an HPC component framework based on commodity models, or to creating bridges between CCA components and other component models, e.g., Web Services [Christensen et al., 2001; Foster et al., 2002].

The specific objectives that have guided the development of the CCA are:

1. **Component Characteristics:** The CCA is used primarily for high-performance components implemented in the Single Program Multiple Data (SPMD) or Multiple Program Multiple Data (MPMD) paradigms. Examples of issues that must be resolved to build applications of such components include the necessity to interact with multiple communicating processes, the coexistence of multiple sophisticated run-time systems, message-passing libraries, threads, and efficient transfers of large data sets.
2. **Heterogeneity:** Whenever technically possible, the CCA must be able to combine within one multi-component application components executing on multiple architectures, implemented in different languages, and using different run-time systems. Furthermore, design priorities must be geared toward addressing the software needs most common in HPC environment; for example, interoperability with languages popular in scientific programming, such as Fortran, C, and C++, should be given priority.
3. **Local and Remote Components:** Components are local if they live in a single application address space (re-

ferred to as *in-process* components in some other component models) and remote otherwise. The interaction between local components should cost no more than a virtual function call; the interaction of remote components must be able to exploit zero-copy protocols and other advantages offered by state of the art networking. Whenever possible local and remote components must be interoperable and be able to change interactions from local to remote seamlessly. The CCA will address the needs of remote components running over a local area network and wide area network; distributed component applications must be able to satisfy real-time constraints and interact with diverse supercomputing schedulers.

4. **Integration:** The integration of components must be as smooth as possible. In general it should not be necessary to develop a component specially to integrate with a particular framework, or to rewrite an existing component substantially.
5. **High-Performance:** It is essential that the set of standard features contain mechanisms for supporting high-performance interactions; whenever possible applications should be able to avoid extra copies, extra communication, and synchronization, as well as to encourage efficient implementations, such as parallel data transfers. The CCA should not impose a particular parallel programming model on users, but rather allow users to continue using the programming models with which they are most familiar and comfortable.
6. **Openness and Simplicity:** The CCA specification should be open, and it should be usable with open software. In HPC this flexibility is needed to keep pace with the ever-changing demands of the scientific programming world. Related and possibly more important is simplicity. For the target audience of computational scientists, computer science is not a primary concern. An HPC component architecture must be simple to adopt, use, and reuse; otherwise, satisfying any other objectives will be moot.

3.2 CCA Concepts

The central task of the CCA Forum (<http://www.cca-forum.org>) is the development of a component model specification that satisfies the objectives above. The specification defines the rights, responsibilities, and relationships among the various elements of the CCA's component model as well as the interfaces and methods that control their interactions. Briefly, the elements of the CCA model are as follows:

- **Components** are units of software functionality that can be composed together to form applications. Components encapsulate much of the complexity of the software inside a black box and expose only well-defined interfaces to other components.
- **Ports** are the abstract interfaces through which components interact. Specifically, CCA ports provide procedural interfaces that can be thought of as a class or an interface in object-oriented languages, or a collection of subroutines, or a module in a language such as Fortran 90. Components may provide ports, meaning they implement the functionality expressed in the port (called *provides ports*), or they may use ports, meaning they make calls on that port provided by another component (called *uses ports*). It is important to recognize that the CCA working group does not claim responsibility for defining all possible ports. It is hoped that the most important ports will be defined by domain computational scientists and standardized by common consent or de facto use.
- **Frameworks** manage CCA components as they are assembled into applications and executed. The framework is responsible for connecting *uses* and *provides* ports without exposing the components' implementation details. The framework also provides a small set of standard services that are available to all components. In order to reuse concepts within the CCA, services are cast as ports that are available to all components at all times.

3.3 Overview of the CCA Specification

Formally, the CCA specification is expressed as a set of abstract interfaces [CCA Forum, 2003] in the Scientific Interface Definition Language (SIDL) [Dahlgren et al., 2004, Chapter 3] used by the Babel language interoperability tool (discussed in depth in Section 4), which implicitly defines bindings to the various languages Babel supports. The current CCA specification is version 0.6.3, and it uses the SIDL namespace `gov.cca`.

Central to the CCA specification is the `Services` interface, which is the primary means by which components interact with the framework, allowing the component to inform the framework of port interfaces that the component intends to use or provide. The `Services` object also permits the component to avail itself of other services that the framework may provide, such as information about connections between itself and other components, or the ability to instantiate and otherwise manipulate other components.

`Services` provides methods to declare what ports the component will *provide* and *use* (`addProvidesPort()`, `removeProvidesPort()`, `registerUsesPort()`, `unregisterUsesPort()`), making it possible for the

CCA framework to effectively mediate port connections in applications. These methods are most often used when the component is instantiated by the framework, but can also be used at other times. `Services` also includes the methods that allow a user to get a handle to a port in order to make method calls on it (`getPort()`, `getPortNonblocking()`, `releasePort()`). These latter methods are generally used in the main code of a component, bracketing regions of code where calls are made to methods on a port provided by another component. A recent addition to the `Services` interface is the ability to register a callback that the framework will invoke immediately prior to destroying the component (`registerForRelease()`). This capability provides the component an opportunity perform any special cleanup operations that might be necessary, though in practice, we find that relatively few components need to avail themselves of this capability and its use is completely optional.

This `Services` mechanism differs from other component models (e.g., CORBA CCM[Object Management Group, 2002]) that require components to implement all functionality of the component model within the component itself. In the CCA, we have begun simply and augment the `Services` specification with more methods as the need arises. Since `Services` is the responsibility of the framework, extending the `Services` interface with new methods does not break existing component code. Components can later take advantage of new features as part of their normal development process.

To be a CCA-compliant component, it is necessary to implement the CCA's `Component` interface, which specifies just one method, `setServices()`. When the component is instantiated in a CCA framework, the first thing the framework does is invoke the component's `setServices()` method, passing in an object of type `Services`, and this generally is where the component registers the ports it *uses* and *provides*. This definition of the `Component` interface is part of the CCA's minimalist approach – the number of new methods required to become a CCA component (sometimes referred to as *surface area* [Szyperski, 1999]) is as small as possible: exactly one.

At the user level, components are both typed and given individual identifiers. Both are strings, with the type being conventionally the SIDL class name of the component, and the identifier being at the discretion of the user assembling the application, with the requirement that it be unique within the application. Internally to the CCA framework, components are identified by unique opaque component IDs, which can be obtained through the `ComponentID` interface.

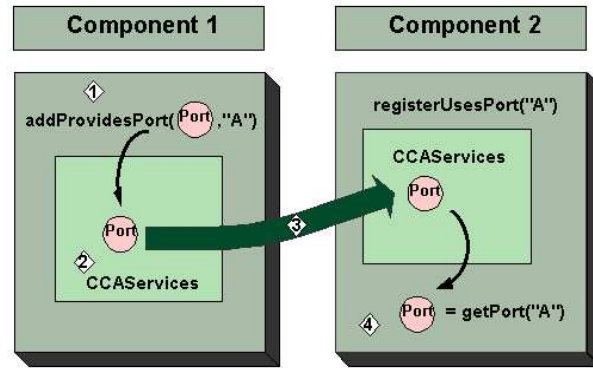
As with components, CCA ports are identified by a type and name, both strings. By convention, the type is set to

the SIDL interface name of the port, and the name must be unique within the component. The functions that register ports also accept a `TypeMap` (a map containing key/value pairs of any basic SIDL data type) specifying properties for the port. The `Services` interface also includes a `getPortProperties()` method to retrieve this information. To implement a CCA port, it must inherit from (or extend) `Port`, which serves only as a container and has no required methods. Components may provide multiple ports, and even multiple instances of the same port.

In addition to defining the general CCA port mechanism, the CCA specification also defines a number of specific ports. `GoPort`, `BasicParameterPort`, and `ParameterPortFactory` are specified as conveniences for component writers, to standardize some basic operations. The `GoPort`, with its `go()` method (with no arguments), is a specific type of port that frameworks and application assembly GUIs can recognize as a means to initiate an operation in the component (i.e., start the application). The other two ports facilitate the implementation of user- and component-accessible interfaces to set input/control parameters that components might need to operate.

The CCA reuses this port mechanism to export services provided by the framework. The only difference between a framework-provided service port and a port provided from a connected component is availability: the framework service port is always available, while the port from a connected component can be obtained only after a connection has been made. CCA-defined ports, such as `ConnectionEventService`, `BuilderService`, `AbstractFramework`, and `ComponentRepository` are services that all frameworks are required. `ConnectionEventService` provides a means for components to find out when connections are made and broken. `ComponentRepository` is currently a placeholder for a planned interface that will allow component implementations to be retrieved from distributed repositories. `BuilderService` and `AbstractFramework` provide a means to programmatically assemble and modify applications (instantiate and destroy components, make and break connections between ports), and the means for arbitrary code to become a CCA framework. These services not only provide the basis for application builder GUIs to be easily constructed, but also allow dynamic behavior of the application itself, for example, swapping components based on numerical or computational performance [Hovland et al., 2003; Norris et al., 2004; Bernholdt et al., 2004]. `BuilderService` also enables encapsulation of assemblies of components so that they can be treated as a single component. This capability facilitates managing the complexity of large component-based applications and also facilitates the assembly of multi-scale or multi-physics simulations, where complex applications representing a particular length scale or type of physics can be encapsulated and treated as a single component, exposing only a limited number

Figure 1: A schematic representation of how ports are connected and used through a sequence of interactions between a component and framework via the CCA Services object.



of ports.

Exactly which ports and components are connected and in what way is mediated by the framework (see below), which in turn is controlled by a script or interactively by the user. Because ports are nothing more than interfaces, components that reside in the same process — as they almost always do in the HPC case — have direct access to methods, just as one would in a scientific subroutine library. Except for usually negligible overhead from the presence of virtual function calls, directly connected components have the same performance as their non-componentized counterparts. This means that once the component composition phase of an application is finished, an application created from CCA components has performance characteristics that are little different from that of non-component code.

3.4 Component Lifecycle

Figure 1 illustrates more specifically how ports are connected and used through a sequence of interactions between a component and framework via the CCA Services object. In Step 1, Component 1 calls `addProvidesPort()` (and Component 2 calls `registerUsesPort()`) to express their intent. Then in step 2, the CCA Services object caches the information about the port that it received from `addProvidesPort()`. In the third step, the framework connects the *uses* port to the *provides* port, and the framework copies information about the *provides* port to the user's (Component 2's) CCA Services object. Finally, when Component 2 wants to invoke a method on the port provided by Component 1, it issues a `getPort()` call to obtain a handle to the port. Not shown in the diagram is the `releasePort()` call, which informs the framework that the caller is (temporarily) done using the port. A port may be used only *after* a `getPort()` call has been made for it, and *before* its companion `releasePort()` call; `getPort()` and `releasePort()` can be used repeatedly throughout the body of the component. This approach

is considered better CCA programming practice than acquiring handles to all relevant ports once at the beginning of the component execution and releasing them only at the end, as it allows the use of a more dynamic component programming model.

Finally, when not within a `getPort()/releasePort()` block, connections between *uses* and *provides* ports may be broken and reconnected, and components may be destroyed. In general, components cannot use ports on other components during the composition phase (e.g., within the component's `setServices()` routine) because there is no guarantee that the components providing those ports have been instantiated and connected to this component's *uses* port. Framework service ports are always available and can be used at any time.

The framework and the CCA model do *not* require that all *uses* ports declared by a component be connected to *provides* ports. While there are many cases in which this is an error (akin to a traditional application not being linked to a library it needs), it also provides a useful flexibility to the component developer. For example, needed functionality might be provided by any of several different ports, and this approach allows the component to check which are available (i.e., connected) and use the best one. Or some *uses* ports might be considered optional by the component developer – to be used if present, but if not, the component should continue without failing. Examples might include an optimization component that will make use of the function's Hessian (second derivatives) if connected to a port providing it, but will otherwise use an internally computed approximate Hessian, or a component that will send its text output to another component if connected (perhaps one that collects and annotates the output of a parallel application with information about the originating process) but simply will use `stdout` if not connected.

In everyday use of the CCA model, steps 1–3 above would take place during the composition phase of the applications. Specifically, steps 1 and 2 would take place with the component's `setServices()`, invoked by the framework when the component is instantiated, and step 3 would take place as the component instructs the framework how to connect the *uses* and *provides* ports for the application. Step 4 would take place during execution of the component's code.

While this explanation has portrayed the phases of the lifecycle as *collective*, with the entire application being assembled, executed, and then disassembled, this is not necessarily the case. As mentioned above, the `BuilderService` framework service port allows all lifecycle operations to be done under the programmatic control of any component. This behavior allows the component to proxy the behavior of the framework (e.g., to recover the concept of a “main”

program, which otherwise is not part of a general component model like the CCA) but is considered advanced use. The developer of such a component must be keenly aware of the CCA rules and practices governing component behavior to ensure that all components in the application “see” the expected CCA environment.

4 Language Interoperability

Multi-language programming is a fact of life in scientific computing, making programming language interoperability an important issue that the CCA is addressing. Scientific applications often contain combinations of compiled languages, such as C, C++, Fortran 77 and Fortran 90/95, and scripting languages, such as Perl, Python and Tcl. An informal survey of scientific coding groups, mostly from DOE labs, found that the average project used more than three computer languages [Kumfert and Epperly, 2002]. Furthermore, as the community builds more CCA-compliant components, from scratch and legacy codes, there is every reason to expect the trend toward multi-language applications to continue to grow.

Unfortunately, the task of getting codes written in different programming and scripting languages to interoperate can be quite burdensome if there is no native support in one or more of the languages. Historically, developers have had to write code by hand to “glue” multi-language applications together, most commonly in a pairwise fashion. As a result, changes that impacted the interfaces between the languages had to be reflected in changes to the glue code. As the number n of supported languages increased, the language interoperability burden increased n^2 .

This maintenance nightmare has led to a number of efforts to ameliorate the problem. For example, Simplified Wrapper and Interface Generator (SWIG) [Beazley, 2003] is a tool that enables interactions among a select group of languages. SWIG allows code written in Perl, Python, Ruby and Tcl to call software written in C or C++, but it does not enable C/C++ to call Perl, Python, Ruby or Tcl.

Rather than rely on the current norm of pairwise interoperability between a few languages, the CCA has adopted the use of a tool that reduces the language interoperability burden on component developers and users. This tool, called *Babel* [Lawrence Livermore National Laboratory, 2004], reduces the burden from n^2 to $2n$. Babel is able to substantially reduce the interoperability effort through the use of an intermediate object representation (IOR) implemented in ANSI C [Dahlgren et al., 2004]. Currently, Babel supports C, C++, Fortran 77, Fortran 90/95, and Python. Actually Babel is a “standalone” tool in that it can be used outside of the Common Component Architecture environment.

Babel is an interface definition language (IDL)-based tool that automatically generates code to glue multi-language components together. It relies on the Scientific Interface Definition Language (SIDL) [Dahlgren et al., 2004] for the definitions of calling interfaces through defined types (i.e., interfaces and classes) and declared methods. A SIDL *interface* is a collection of methods that cannot be implemented directly. A SIDL *class* is also a collection of methods which, in this case, can be implemented so that associated objects can be created. An exception is made for abstract classes whereby one or more of the methods are not implemented so objects of the class cannot be created. Interfaces and classes are grouped together in packages to provide a namespace for symbol name collision prevention. Using a SIDL file, the Babel compiler generates the glue to enable caller and callee to be in any supported language. The generated glue code handles the translation of arguments and method calls between languages.

Like similar technologies such as CORBA [Object Management Group, 2002] and COM [Microsoft Corporation, 1999], Babel provides the standard hallmarks of object-oriented programming, such as object identity, inheritance, polymorphism, and encapsulation, in all supported languages even when the features are not native to the language. In fact, Babel's inheritance model is analogous to that of Java [Gosling et al., 1996]. A SIDL class can extend, or inherit from, at most one class and can implement an arbitrary number of interfaces. Single inheritance of classes was chosen to avoid the confusion of multiple inheritance. Interfaces can extend multiple interfaces. With Babel, it is possible to implement each level of a type hierarchy in a different language. This is inherent in Babel's object model because all classes specified in SIDL, regardless of their language of implementation, implicitly extend the SIDL base class, `sidl.BaseClass`, which is implemented in C.

What makes Babel and SIDL different from other IDL-based tools, such as CORBA and COM, is that they are being tailored for use in the scientific computing community. For example, SIDL has built-in support for scientific data types, including complex types and dynamically allocated, multi-dimensional, arbitrarily strided arrays.

Furthermore, having the IOR implemented in ANSI C has several benefits of importance to the community. First, it enables the generation of glue code that will support fast, single-process communication, which is important to many of today's applications. Second, using ANSI C increases the portability of the glue code across high-end platforms. Finally, Babel supports Fortran as a first class language. This is important because Fortran is largely ignored by most of the computing world these days, including most of the commodity component models. However, it has a large following in scientific computing, particularly among application developers. Babel provides native Fortran 90/95

array access to Babel's internal array data structure using the CHASM array descriptor library [Rasmussen et al., 2003; Rasmussen et al., 2001]. Providing native access to array descriptors is challenging because array descriptor internals are usually compiler specific and undocumented.

Since component technologies are an evolutionary step beyond object-oriented programming, the CCA has been able to leverage Babel and SIDL in the development of its component framework. The CCA specification is written in SIDL, and component developers write SIDL files to describe their ports and the classes that use or provide those ports. Using SIDL enables the encapsulation of implementation details of CCA-compliant components. Wrapping the CCA framework in SIDL makes it a relatively simple exercise to create Python components serving as application drivers, GUIs, and other tasks for which scripting languages are often used in traditional programming environments [Kumfert, 2003].

Benefits of Babel technologies include its environment, programming language support scalability, and potential as a basis for future research. Babel is a portable, language independent, object-oriented programming environment that enables the CCA to incorporate components written in the most common scientific programming languages. Furthermore, it provides a scalable approach to support new languages as they rise to prominence. Other advantages include the ability to leverage the tools themselves to facilitate scientific computing research in areas such as remote method invocation, specification-level parallelism, and dynamic code insertion. More information about current efforts on these research areas can be found in Section 15.

5 Fine-Grained Component Interactions

As mentioned previously, one of the major requirements of the Common Component Architecture is that it impose minimal performance overheads on both local component interactions and component-based parallel applications. While not explicit in the CCA specification, the design of the CCA makes it possible for straightforward implementations of the specification to obtain high performance in both contexts.

In the CCA specification, the framework mediates all connections between *uses* and *provides* ports. This approach makes it possible for CCA implementations to transparently support both distributed and local connection models. In the distributed case, the `getPort()` invocation would return a pointer to a proxy for the *provides* port provided locally by the framework, and the framework itself would be responsible for conveying the method invocations to the

actual remote port (see Section 7).

In the local case, CCA implementations generally load components into the same *process* so that they share the same memory space, while putting them into different *namespaces* to preserve the separations expected between components. In this case, `getPort()` can simply return a pointer to the virtual function table for the port, thereby allowing methods on the port to be invoked directly (without further intervention by the framework), and allowing data to be passed by reference if desired. The CCA term for this approach is *direct connection*. In component models, most of which were designed primarily with distributed computing in mind, direct connection optimizations would violate the component model's specifications. Some CORBA ORBs provide a less efficient collocation optimization, which does maintain compatibility with the component model [Schmidt et al., 1999].

Experiments have shown that the overheads for calls between directly connected CCA components are small and quite manageable in the context of scientific computing [Bernholdt et al., 2002b; Norris et al., 2002; Benson et al., 2003a]. Calls using simple data types (those requiring no translation by Babel) cost the equivalent of a C++ virtual function call, which is exactly what is to be expected for the direct connection implementation described above. For those unfamiliar with C++, this is roughly three times the cost of a function call in Fortran or C, or about 50 ns on a 500 MHz Pentium III system. For comparison, the same test with the omniORB CORBA implementation on the same platform required approximately twenty-six times longer.

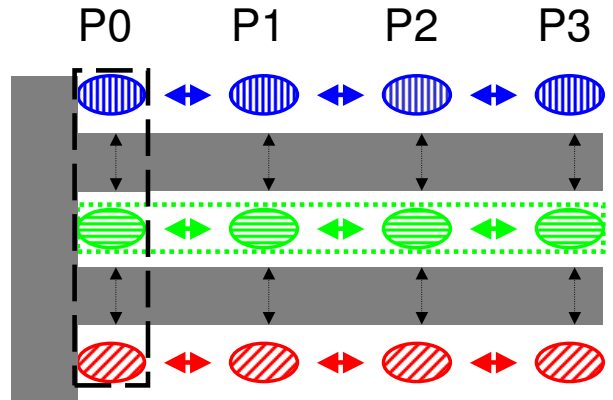
Of course there are data types that require more effort on Babel's part to translate from one language to another – effort that varies depending on the two languages being connected. For example, nearly every language represents strings differently, so it is generally necessary to allocate memory and copy for every string argument. Babel also allows the user to specify that arrays always be in a fixed order (row- or column-major), and between some language pairs must allocate and copy the entire array to maintain the specified ordering. However, for general scientific computing, it is not difficult to avoid most overheads associated with componentization in the CCA model, or to amortize them by doing a sufficient amount of computational work for each function invocation. The advice given to developers of CCA-based applications is to be aware of the costs and factor them into the overall design of the application and the design of the individual components and interfaces.

6 Parallel Components

The CCA’s approach to supporting high-performance parallel computing is guided primarily by the “Integration” objective of Section 3.1. The simplest approach to integrating existing parallel code in a CCA environment is to accept whatever parallel programming model was used in the original code, rather than trying to impose a new CCA-specific one. This approach offers a number of additional benefits. First and foremost, software developers can continue to take advantage of the investment they have made in learning how to produce high-performance software in their favorite programming model. We believe that for many users, this capability is as important as the more obvious fact that our approach requires a minimal modification to existing code. In addition, the CCA does not require yet another parallel programming model, which greatly simplifies the task of implementing a CCA-compliant environment.

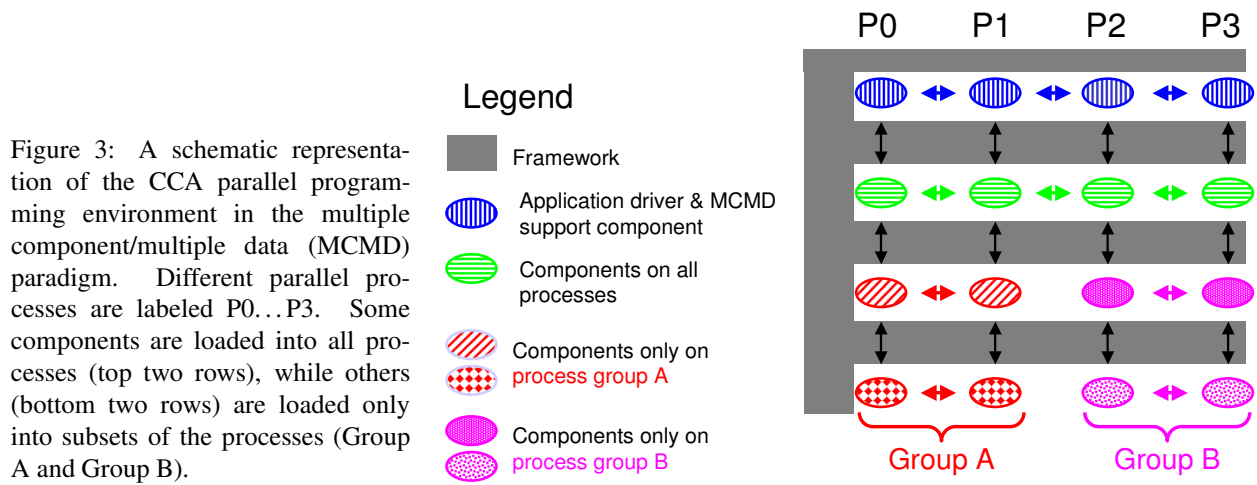
There are a variety of CCA framework implementations that vary in their support for parallel and distributed computing. At present, a given framework supports *either* HPC parallel *or* distributed computing, and parallel plus distributed applications must be assembled from pieces running in different frameworks, using the `BuilderService` and `AbstractFramework` services defined in the CCA specification. As discussed further below, we expect to move toward higher degrees of integration and transparency in the coupling of parallel and distributed applications. For now, however, we focus on the current situation with respect to parallel and distributed computing.

Figure 2: A schematic representation of the CCA parallel programming environment in the single component/multiple data (SCMD) paradigm. Different parallel processes are labeled P0...P3, and the same set of three components has been loaded into each process. Components in the same process (vertical dashed box) interact using standard CCA port-based mechanisms, while parallel components of the same type (horizontal dotted box) interact using their preferred parallel programming model, such as MPI, PVM, Global Arrays, etc.



Ccaffeine [Allan et al., 2003; Allan et al., 2002], the main CCA framework implementation for HPC parallel computing, supports the component analogs of both the single program/multiple data (SPMD) and multiple program/multiple data (MPMD) models, to which we refer as single or multiple *component*/multiple data (SCMD, MCMD) models. Figure 2 depicts the SCMD case: each process is loaded with the same set of components wired together in the same way. Interactions among components within a given process (vertical direction) take place through

the normal CCA means — by declaring used and provided ports and using `getPort()`/`releasePort()` around calls on other ports. Interactions across a component’s parallel *cohort* (parallel instances of the same component, the horizontal direction in the figure) take place via whatever parallel programming model that component wishes to use. Different components within the same application may use different parallel programming models (with certain caveats, discussed below). “Diagonal” interactions (between component A on one process and component B on another process) are not prohibited by the CCA, but require a degree of coordination that may not be consistent with the notional independence of components (unless a componentization has been carried to a degree that groups of closely intertwined components are used to carry out certain operations).



In the MCMD case (Figure 3), some components are typically loaded into all processes, while others are loaded only into subsets of the processes. An example of an MCMD application might be a coupled global climate model, where atmospheric, oceanic, land, and other elements of the model are run simultaneously on different subsets of the available processes. These separate elements are managed by a driver component and supported by other components to handle the data exchange between the various elements. In contrast to the atmospheric, oceanic, and other elements, the management and support components are loaded into all processes of the simulation. Because of the complexity of partitioning up the process space and of launching parallel jobs with different inputs or executables on every process, MCMD applications can perhaps be most easily created through the use of the CCA’s `BuilderService` framework service from a so-called builder component. This component, which might conveniently be written in a scripting language, such as Python, would compute the desired partition of the available processes and then use `BuilderService` on each process to load and connect the appropriate set of components. The builder would then

end by invoking the `GoPort` on the climate model’s driver component to initiate the simulation.

Ccaffeine also supports threaded parallelism, with the caveat that on each process, only the main thread is allowed to interact with the framework to manage connections between components. The SCIRun2 framework [Zhang et al., 2004] is fully thread-safe, thus allowing combinations of MPI-based and thread-based parallelism within the same component. However, neither of the systems yet provides mechanisms for managing use of resources in an environment where more than one multi-threaded component is competing for processing resources.

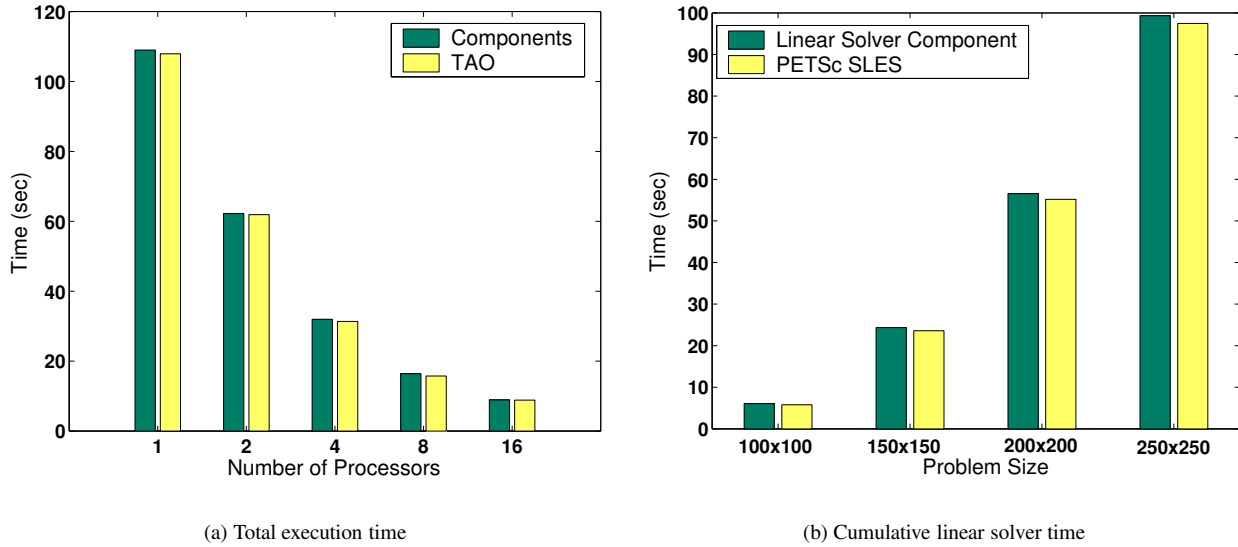


Figure 4: Component overhead in an unconstrained minimization application on a 250×250 mesh: (a) total nonlinear solution time (b) linear solution time.

The simplicity of the CCA approach to parallel computing offers another significant advantage besides the ease of incorporating existing parallel software into the CCA environment: the performance of parallel components is virtually identical to that of the original code. The only overheads imposed by the CCA model of parallelism are those on the local component interactions, which, as discussed in the previous section, are small, and quite manageable. A number of studies have verified this experimentally:

- Norris and coworkers [Norris et al., 2002] have compared a CCA component implementation and a non-component library implementation of a nonlinear unconstrained minimization problem solved using an inexact Newton method with a line search [Norris et al., 2002]. Each iteration of the Newton method requires function, gradient, and Hessian evaluations, as well as an approximate solution of a linear system of equations. The over-

head of the component implementation results from the extra layer of abstraction (more virtual function calls) and is most significant for very small problem sizes. The parallel performance results on 1 to 16 processors of a Linux cluster also demonstrate that the SPMD component implementation does not adversely affect parallel performance and scalability. Figure 4 compares the component and non-component costs for the overall minimization simulation and the linear solve phase.

- Katz, Tisdale and Norton [Katz et al., 2002] constructed a simple component-based two-dimensional parallel unstructured Adaptive Mesh Refinement (AMR) application and compared it with the library on which the component was based. Repeated timings (5–10 runs each) on between 2 and 32 processes showed no effect on the parallel scalability of the application and essentially no difference in run times (some runs showed the library version slightly faster, while for others the component version was slightly faster).
- Benson et al. [Benson et al., 2003a] analyzed the performance of a molecular optimization problem [Hoare, 1979] similar to the one described in Section 13.3. This application used a two-dimensional Lennard-Jones potential together with the Toolkit for Advanced Optimization (TAO) [Benson et al., 2001; Benson et al., 2003b] for the optimization and Global Arrays (GA) [Nieplocha et al., 1996; Pacific Northwest National Laboratory, 2004a] for the linear algebra. Tests were performed with up to 65,536 atoms on up to 170 processors to evaluate the overhead of the CCA component version with respect to the non-component library-based version of this application, and the largest overhead found was less than 0.2% of the total execution time (Figure 5).

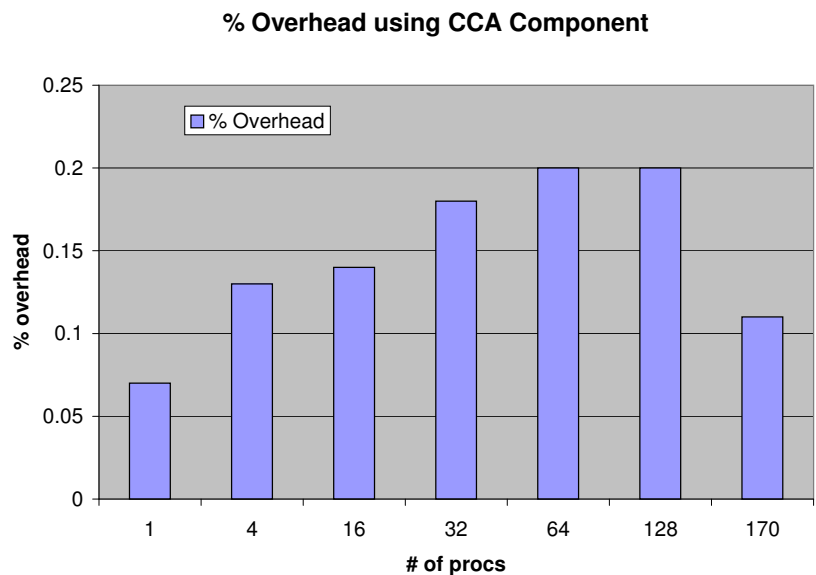


Figure 5: Measured overhead as a percentage of execution time for component- vs. library-based implementations of a Lennard-Jones energy optimization problem.]

The simplicity of the CCA approach to parallel computing also brings with it some issues and possible disadvantages that must be mentioned. First, the programming models (and specific implementations) used in the various components assembled into an application must be “composable”, or able to work together without interference. In many cases, this issue does not really pose a problem. For example, MPI [Forum, 1994], PVM [Geist et al., 1994], and the Global Array Toolkit [Pacific Northwest National Laboratory, 2004a; Nieplocha et al., 1996] can coexist together without problems thanks to their compatible execution environments. On the other hand, connecting an MPI-based component to one based on OpenMP is unlikely to produce the desired result without careful thought, and perhaps additional programming. While the composition of parallel programming models in a single application is perhaps more likely to occur in a component environment than in traditional application development, this issue is in no way specific to component environments and is outside the scope of the CCA.

In addition, although SIDL bindings exist for MPI, PVM, and Global Arrays, no parallel programming model has yet been formally componentized in the CCA environment. This situation means that they must be treated as traditional libraries, which raises certain subtle issues in a component environment. Basically, whether the parallel programming (or other) library is linked to the component instead of the framework itself, and whether the linkage is static or dynamic, affect whether each component sees the same parallel environment or a completely separate instance of it [Bernholdt et al., 2002a]. In most cases, the desired result of a shared parallel environment requires that the library be linked into the CCA framework, which is a minor inconvenience (in so far as the framework build must be modified to include the libraries required by the application(s) of interest), and begs the question of who takes responsibility for initializing and finalizing such libraries. The present solution is that the framework is also responsible for these actions, which has proven adequate so far. In principle, they could be moved into a component, but MPI, for example, requires the application’s command line arguments for its initialization procedure, which in turn suggests the need for a standardized means by which components can obtain from the framework a copy of the commandline arguments.

Finally, for any programming model that provides support for multiple contexts, code is much more flexible and much more easily reused when written to accept an externally determined context rather than use the default global context (such as MPI’s `COMM_WORLD`). This is advice often given to those programming in message-passing environments (and just as often ignored, it seems), which can become all the more important in component environments, particularly for components that might be used in the MCMD fashion.

7 Distributed Components

Although historically high-performance scientific computing has focused primarily on local sequential and parallel computing, there is an increasing interest in distributed computing, shadowing the rise of the Internet, web-based “application service providers”, and Grid computing [Foster and Kesselman, 1998]. The CCA’s design objectives (Section 3.1) include support for both parallel and distributed computing approaches. The primary distinction between the two paradigms, as far as the CCA is concerned, is whether or not it is possible for components to be directly connected, sharing memory within the same process. As described above, such direct connection is the primary means by which CCA environments can assure the highest possible local/parallel performance; in distributed computing, the emphasis is less on performance (though efficient protocols are still important) than on flexibility and the ability to integrate with existing distributed computing standards and environments. Beyond this, the integration of the HPC parallel and distributed paradigms within a single component-based environment is an interesting and challenging research issue.

Distributed computing environments raise a number of issues that differ from those in the direct-connect parallel paradigm:

- **Component Interaction:** When components cannot be directly connected because they are in separate processes, generally in disjoint address spaces, it is necessary for the framework to mediate calls on ports through mechanisms such as remote method invocation (RMI) [Sun Microsystems, 2004b], remote procedure calls (RPC) [Birrell and Nelson, 1984], or message passing [Forum, 1994]. Numerous such systems are available (and new ones continually being developed), which offer a range of performance and capability. In each system, there is a specialized representation for “handles” to the ports and services of components which must be designed as globally unique references to point to objects in remote address spaces. Often these handles are designed as objects that can function as proxies for remote objects.
- **Registration and Discovery:** This is the term for the mechanisms that allow components to make their availability known and to find each other so that they can cooperate within the same application. Though this capability is necessary for both local/parallel and distributed computing, in the former case, very simple solutions are quite effective: components are typically loaded from pre-specified locations on a (shared) file system into a single

framework instance. In the distributed case, one cannot count on shared file systems, or even ones that have similar structure, and the CCA framework itself must execute in distributed fashion, so that component co-location cannot be the basis of discovery. Registration and discovery (R&D) mechanisms are very much influenced by the integration of the CCA into broader distributed computing environments and the mechanisms they use for R&D.

For example, in an Open Grid Services Architecture (OGSA) [Foster et al., 2002] or Web Services-based environments, components can be described by Web Services Description Language (WSDL) documents [Christensen et al., 2001], and registered and discovered in registries such as the Lightweight Directory Access Protocol (LDAP) [OpenLDAP Foundation, 2003] and Universal Description, Discovery, and Integration (UDDI) [UDDI.org, 2003]. For Legion-based environments [Grimshaw et al., 1999; Lewis et al., 2003], components have names in a globally accessible context-space, and can be discovered and filtered based on where they are running, to which class they belong, their creator, and a range of other component attributes.

- **Component Creation:** In a distributed environment, the creation of components needed for an application typically involves remote resources, whereas in a local/parallel situation the resources are local, and typically pre-allocated (for example by a queuing system). Distributed frameworks must therefore support dynamic remote creation and placement (i.e., scheduling) that respects the security (authentication and authorization) requirements of the component, the application, and the supporting environment.

Given the ranges of issues and options for addressing them, including choices of a number of well-established distributed computing environments into which CCA's CBSE approach might be integrated, it will not be surprising that there are currently several distributed CCA frameworks supporting a range of capabilities.

7.1 XCAT

XCAT (version 2.0) [Govindaraju et al., 2003] is a CCA-based distributed component framework that uses the Web Services model as its basic architecture. It has been implemented in both C++ and Java, and provides interoperability between components developed in these two languages. Each *provides* port in an XCAT component is described using an XML schema and is designed as a Web Service that has one port type. The XCAT team is currently working on using a WSDL [Christensen et al., 2001] document for this purpose. WSDL documents will provide a description of

the interface and also provide information pertaining to the communication protocols to which the service is bound. The use of WSDL for XCAT components will enable the use of well known Web service client toolkits to invoke methods on XCAT *provides* ports.

Component creation and launching mechanisms on remote locations are based on Globus GRAM via the Java CoG kit [von Laszewski et al., 2001]. For components in the same address space, XCAT has a in-built mechanism that is optimized for co-location optimization. Communication among components is enabled by the XSOAP [Slominski et al., 2001] toolkit, which provides an elegant model based on the Java-RMI (Remote Method Invocation) specification [Sun Microsystems, 2004b]. The Java based implementation of XSOAP and XCAT use the dynamic proxy feature of Java. This feature allows the generation of stubs and skeletons dynamically.

While the XCAT API can be used directly in programs for composition of applications using the component assembly model, the program must be recompiled whenever the component wiring is modified. XCAT also provides a binding to Jython (an implementation of Python in Java) [Anonymous, 2003] that can be used to access the XCAT API, thereby enabling scientists to change their application composition without the need for recompilation. This approach also has the added advantage of launching applications from web-portals. Recent work with XCAT has also focused on the design of Open Grid Services Infrastructure (OGSI) [Global Grid Forum, 2003] compatible CCA components [Krishnan and Gannon, 2004].

7.2 SCIRun2

SCIRun2 (version 1.90) [Zhang et al., 2004] is a framework that combines CCA compatibility with connections to other commercial and academic component models. Based on the SCIRun[Parker and Johnson, 1995; Johnson and Parker, 1995; Parker et al., 1997] infrastructure and the CCA specification, SCIRun2 utilizes parallel-to-parallel remote method invocation to connect components in a distributed memory environment, and it is multi-threaded to facilitate shared memory programming. SCIRun2 also has an optional visual-programming interface. Overall, this framework provides a broad approach that will allow scientists to combine a variety of tools for solving a particular problem. The overarching design goal of SCIRun2 is to provide the ability for a computational scientist to use the right tool for the right job, a goal motivated by the needs of our biomedical and other scientific users.

The primary innovative design feature of SCIRun2 is a meta-component model that facilitates integration of a

number of classes of tools from various, previously incompatible systems. In the same way that components plug into CCA or other component-based systems, SCIRun2 allows entire component models to be incorporated dynamically. Through this capability, SCIRun2 facilitates the coupling of multiple component models, each of which can bring together a variety of components. We are utilizing this feature to enable the coupling of single-address-space components based on Babel (see Section 4), components from SCIRun, as well as CCA components that use the SCIRun2 distributed computing infrastructure. Special components, called bridges, facilitate interactions between components belonging to different models. These bridges can be automatically or semi-automatically generated.

The SCIRun2 framework enables a variety of distributed computing mechanisms to be used together. Natively, SCIRun2 provides support for RMI-based distributed objects. This support is utilized in the core of the SCIRun2 framework in addition to distributed components. Using the SIDL language, we compile proxy objects that marshal method calls to remote objects and manage distributed references. For objects that exist within the same address space, this marshalling framework is completely bypassed, making fine-grained component interactions possible with only the cost of a C++ virtual function call. For distributed components, the SCIRun2 framework employs a remote *slave framework* that manages component creation, registration and discovery on a remote, possibly parallel, computing resource. We extend the SIDL language and build upon the SCIRun2 RMI system to implement parallel-to-parallel component connections, discussed further in Section 12.

7.3 Legion

Legion is an object-based Grid computing system [Grimshaw et al., 1999; Lewis et al., 2003]. The Legion CCA framework [Lewis et al., 2003] makes use of the Legion object model and run-time system for implementing components. Legion objects contain their own address space, are named by Legion Object Identifiers (LOID's) that have meaning in a global namespace and are instances of class objects, which serve as their managers at runtime. Currently, each CCA component maps to its own Legion object and therefore has its own address space. Components make calls on others in the CCA framework by invoking methods on the Legion objects that represent them. Programmers specify CCA components using SIDL, and a stub generator produces wrapper code that allows each component to communicate with the others in a Legion-based framework. Programmers fill in port definitions and compile the code into a Legion object that can then run in the Legion-CCA framework as a component.

Each component is linked with both the Legion runtime library and a CCALegion library, which delivers the services required by the CCA specification. For example, the library furnishes an interface for registering and obtaining *provides* and *uses* ports, for instantiating components via a BuilderService, and for maintaining a table of connections to other components. Many of the library's functions require outcalls to remote services, but the interface to the component programmer is provided as a local API. This approach keeps programming components that are intended to run in a distributed environment as similar to programming local components as possible.

Legion provides services that cover all of the distributed computing issues raised at the beginning of this section. Thus, implementing a particular service is a matter of mapping the CCA request onto the Legion implementation. For example, component creation is realized by invoking the `createInstance()` function in the Legion library, which in turn calls the `CreateInstance()` method exported by the appropriate class object. Component interaction is implemented on top of the Legion messaging system, which supports remote method invocation. In addition, registry and discovery service, although not currently implemented, will be able to take advantage of Legion context space.

7.4 Distributed Interoperability

Interoperability among distributed frameworks differs significantly from other kinds of interoperability and requires an additional specification that is not a contract between the framework and the component, but rather a contract among frameworks. Some of the capabilities required for interoperability among distributed CCA frameworks include:

- **Common Communication Protocols:** Before two components can communicate, there must be some agreement on the protocols they use. For example, they may choose to use something based on the SOAP [Gudgin et al., 2003] standard being developed by the W3C.
- **Common Remote Reference Format:** To connect a *uses* port to a *provides* port, there must be some way to refer to the *provides* port. To connect a *uses* port from one framework to a *provides* port of another, the ports must have a common reference format.
- **Common Creation Protocol:** The current CCA specification provides a BuilderService to be used for creating components. A `ComponentID` is returned upon successful instantiation of the component. In a distributed

framework, once a component has been instantiated, the BuilderService and the new component need to interact so that the `ComponentID` can be returned. This interaction requires an additional protocol above that used for basic communication.

- **Common Connection Protocol:** When two components are connected, a reference of the *provides* port is placed in the *uses* port of the other component. In a distributed framework, the reference to the *provides* port may belong to a component in a different framework. Different frameworks must agree upon the set of RPC calls and on-the-wire format of parameters that will be needed to place a reference to the *provides* port in the *uses* port. Note that this protocol is layered above the common communication protocols.

Part of the ongoing research activity in the distributed CCA effort is aimed at establishing the full requirements for distributed framework interoperability and determining the most effective way to specify each aspect for inclusion in the formal CCA specification. Rather than “blessing” specific protocols for each capability, we anticipate an environment that allows frameworks to negotiate among multiple supported options for each capability.

8 Current Status of the Common Component Architecture

Although active research and development of the Common Component Architecture continue, a variety of tools are available that provide a highly functional environment for the development and use of CCA-compliant components and applications. Indeed, a number of applications groups have already begun using these tools to develop components and component-based applications, as described above.

Although the CCA Specification, currently at version 0.6.3, continues to evolve and mature, most of the recent changes have either been clarifications or the addition of capabilities that are considered advanced, and therefore rarely employed by typical users. For mainstream use, the CCA specification has been largely stable for the last several years. Backward compatibility of the specification as it evolves is not currently a requirement, although the impact on user code is carefully considered before changes are made. For example, the recently added `ComponentRelease` interface was one of three alternatives proposed to provide the desired capability, and was chosen in large part because it had the least backward compatibility problems.

The Babel language interoperability tool, currently at version 0.9.0, is independent of the CCA Specification, but

Table 1: Summary of versions and capabilities of core CCA tools as of 31 January 2004.

Category	Name	Version	Capabilities
CCA Specification		0.6.3	
Language Interoperability	Babel	0.9.0	Supports C, C++, FORTRAN 77, Fortran 90/95, Python
Language Interoperability	Chasm	1.0.1	Used by Babel for Fortran 90 array support.
Framework	Ccaffeine	0.4.3	Local direct-connect and parallel computing (SCMD and MCMD paradigms)
Framework	XCAT	2.0	Local direct-connect and distributed computing compatible with Grid- and web-services approaches
Framework	Legion-CCA	0.3	Distributed computing compatible with the Legion environment
Framework	SCIRun2	1.90	Local direct-connect, MPI/threaded parallel, and distributed computing

along with the specification, is a major dependency for CCA framework implementations. Babel has been evolving fairly rapidly, including the addition of Fortran 90 support within the last year, but backward incompatible changes are announced in advance and discussed within the user community, often influencing the implementation time line. The Babel team has defined a draft set of criteria for the version 1.0.0 release, which include guarantees of longer-term stability.

The final core tool in the CCA environment is a framework. As we have described, there are a number of CCA-compliant frameworks currently available for use (and others which are primarily research vehicles). The primary frameworks, their versions, and their capabilities are summarized in Table 1. Frameworks depend on specific versions of both the CCA specification and Babel, and consequently tend to evolve at a pace guided by releases of those tools. Because of the complexity of supporting multiple (incompatible) versions of specifications and Babel, the CCA frameworks tend to be closely tied to specific releases of these other tools.

A variety of secondary or more specialized tools for the CCA environment are also available or under development. These include tools which automate much of the relatively mechanical work of setting up the “skeleton” of the files required to create ports and components, graphical “application builder” interfaces which connect to CCA frameworks, and tools which facilitate the “componentization” of existing software.

9 Related Work

As has been discussed, the CCA is a component model specialized to meet the needs of the high-performance scientific computing community. While we are aware of no other effort targeting this particular group, we mention in relation to this work a variety of other component models, as well as other tools and environments that facilitate the development of large-scale scientific software.

When considering other component models, probably the most relevant ones to cite are the commodity component models, particularly CORBA CCM [Object Management Group, 2002]; Microsoft(TM) COM, DCOM and .NET [Box, 1997]; Microsoft(TM) Visual Basic [Liberty, 2003]; JavaBeans [Englander, 1997]; and Enterprise JavaBeans [Roman, 1997]. As a group these models were conceived without high-performance computing in mind and usually restrict the user to a platform that is not conducive to HPC. Java components are restricted to the Java platform, and COM/DCOM/.NET are, as a practical matter, restricted to Microsoft operating systems. Neither of these platforms is often associated with high-performance computing. There have been serious efforts to use CORBA in a high-performance setting with mixed results [Keahey, 1996; Schmidt et al., 1996; Denis et al., 2003; Denis et al., 2001]. Commodity component models usually focus on generality and flexibility, and their designs do not emphasize performance in the sense that an HPC simulation developer would define it. The time scales for commodity component models are more often based on human perceptions and reaction time, which is roughly five orders of magnitude too slow for modern parallel computers. However, it is important to emphasize that the CCA considers long-term compatibility with commodity component models important. Indeed, the design of the CCA owes much to these component models, especially CORBA.

In the HPC scientific computing area, perhaps the most relevant comparisons can be made to the variety of popular domain-specific applications frameworks (see Section 2). Cactus [Allen et al., 2000] predates the CCA and originally grew out of the relativistic astronomy community. To this day, it is still fostered by that community but is being generalized to a more multi-purpose framework. The Earth System Modeling Framework (ESMF) [Killeen et al., 2003] is a similar effort, targetted to global climate simulation (see Section 13.2.2). Both have a predefined data model, specific to their application area, which is shared by all of the participating components. The means of component composition is execution order: each component has a “run” or “exec” method that is called in a sequence of the user’s choosing. In both cases there are specializations to their application area that will not be detailed here. Cactus

and ESMF both rely on an execution engine (sometimes referred to as a “flow executive” [Anonymous, 2004]) that is orchestrated by their respective frameworks, while the CCA relies entirely on port composition; execution order and timing are left to the way components use their ports. A third peer component model, Palm/PRISM [Guilyardi et al., 2002], also relatively new, incorporates both of these ideas. While Palm is a generic component framework, PRISM is a specialization dedicated to climate simulation. These have a concept similar to ports, where components are composed by matching typed data structures that flow between components. In addition, Palm has a flow executive that schedules the activation of components during the calculation.

Beyond peer component models, there are packages that are simply labeled “libraries” or “frameworks”. Hypr [Chow et al., 1999; Falgout et al., 2003], Overture [Brown et al., 1999; Henshaw et al., 2002], PETSc [Balay et al., 1997; Balay et al., 2003], POOMA [Reynders et al., 1996; Reynders et al., 2001], and Trilinos [Heroux et al., 2003; Heroux et al., 2004] are all object-oriented class libraries that do not embrace the peer component methodology. The goal of these software packages is to make HPC program construction and execution easier, often in a specific problem domain, e.g., numerical solution of partial differential equations (PDEs). Their approaches are to provide a variety of numerical method implementations (Hypr, PETSc, Trilinos) or to define high-level abstractions for solving a particular kind of problem (Overture, POOMA). None of these packages aims to provide a specification for a general-purpose peer component model.

10 Using the CCA

Although the CCA effort is, in many respects, rather young, the CCA specification and basic tools associated with the CCA environment are sufficiently stable and robust that a wide range of groups are already beginning to use the Common Component Architecture in their software development efforts. The resulting activities range from the creation of components and interfaces encapsulating important capabilities for scientific computing, or for particular scientific domains, to the development of complete component-based scientific applications.

The development of individual components and interfaces for scientific computing is an important activity because it enriches the pool of available components upon which others may draw to assemble their applications. These activities are often carried out in conjunction with the development of a specific application, where developers recognize that particular components may be of broader interest, or by those associated with existing numerical or scientific

libraries who wish to make their tools available in the CCA environment. CCA applications are typically built from a combination of more general, reusable components together with some constructed specifically for the application in question.

It is impossible to convey here the full breadth of work already in progress which utilizes the CCA environment; however, since the ultimate goal of the CCA effort is to change the way scientific software is developed, this overview of the Common Component Architecture would be incomplete if it did not at least present an overview of how the CCA is being used.

In the remainder of this section, we describe the general process of adapting existing code into the CCA environment, or of architecting a “from scratch” component-based application. Section 11 gives an overview of some of the interface and standalone component development activities, followed in Section 12 by a more in-depth description of an important CCA research activity to facilitate the development of coupled simulations. Finally, in Section 13 we sample the diversity of CCA-based application development.

10.1 Architecting Component-Based Software

Scientific simulation codes have traditionally been expert solutions combining a set of numerical algorithms, and physical and mathematical models to solve a specific set of problems. In traditional scientific software development approaches, these models and algorithms are integrated under a common design and generally tightly integrated using common data structures. Their identification as “modules” in an overall solution strategy, however, is typically straightforward.

Whether component-based or not, the natural decomposition of an application is often recursive in nature and intuitive to practitioners in the field. For example, in a multi-physics code, such as global climate simulation, disparate physics are governed by separate sets of partial differential equations (PDEs), and the first stage of decomposition is typically to distinguish among the different physics (i.e., atmosphere, ocean, etc.). These are further decomposed based on the general numerical procedures required: solvers, domain discretizations, physics subassemblies, etc. A third step of decomposition of such a problem typically involves implementing the details of the specific equations (in physics modules), mathematical processes, or numerical algorithms. Pervasive data structures and operations associated with them can often be encapsulated into data objects (in the sense of object-oriented programming) used by many modules

of the application.

Componentization might naturally occur at any of these levels of decomposition (and at different levels in different parts of the application), and may evolve as the application architects and developers modify their requirements, find new opportunities for code reuse, or wish to generalize parts of the code. A coarse-grain decomposition has the attraction of simplicity in defining interfaces and interactions and is often the first step in componentizing an application. However, this approach tends to offer fewer opportunities for code reuse because implementation details (the third stage of decomposition, above) are typically mixed into the higher-level science drivers. Finer-grain decompositions allow for better separation of details of specific physics equations and numerical algorithm choices, making them more amenable to reuse and easy replacement within the application. However, what one gains in flexibility and reusability, one may lose in terms of simplicity. Such approaches require sophisticated users who understand the tools individually as well as their interactions with other tools. This dilemma is not unique to the CCA. Other component-like toolkits (e.g., AVS, Matlab, LabView) present similar issues.

In architecting a component-based application, the interfaces between components become a key element of the design. Interface design is in many respects a social issue – a primary point of cooperation among developers. Depending on the requirements, interfaces may be designed with varying degrees of generality, ranging from being specific to a group of components or a particular application, to being useful across a range of applications developed by multiple research groups and possibly across multiple scientific domains. Clearly the level of effort required to develop an interface scales with its intended generality. However, as is the case with some widely used numerical libraries (i.e., BLAS, LAPACK, and others), the payoff can be high in terms of overall productivity for a more general interface capable of supporting many applications and a variety of implementations. Another aspect of interface design, especially when componentizing existing code, is the opportunity to redesign the interface. In most cases, an interface to a CCA component can be a straightforward translation of the original software's interface in to SIDL. However, some researchers find that the componentization step is an opportunity to redesign the interface based on experience with the original, and use a somewhat thicker wrapper to adapt the new interface to the existing code within the component. It is also important to realize that different aspects of a simulation may naturally lend themselves to different ways of accessing the functionality of a component. This can be provided by a component having multiple ports implementing related interfaces. For example, a component that manages unstructured mesh data may have a port for entity-based

access to the mesh and another for array-based access, with the former providing full capability but lower performance due to the generality of the data model used, while the latter would provide high-performance access to more limited functionality.

10.2 Componentizing Existing Software

Once the component architecture has been determined and the interfaces among components specified, it is necessary to implement the components themselves. The CCA specification is designed to make it as easy as possible to adapt exiting code into components. For most scientific codes, the basic strategy is as follows:

1. Write the wrappers that adapt the source code generated from the SIDL interface definitions to the interfaces in the existing code. The complexity of this step depends on how close the SIDL interface is to the original.
2. Add the code required to interact with the CCA framework, specifically, the `setServices()` method, by which the component tells the framework what ports it provides and uses. If the component requires a signal from the framework that it is about to be destroyed, so that it can perform cleanup operations, the `ComponentRelease` interface must also be implemented.
3. Modify existing tightly coupled implementations containing direct method calls to access the same functionality through ports. This involves adding `getPort()` and `releasePort()` calls and modifying the original method call to use the port handle returned by `getPort()` and comply with the new interface. Some adaptation of internal data objects may be necessary in order to use them as arguments in calls on other ports (for example, a native array being adapted to a SIDL array via the `borrow()` call). As with the wrappers in step one, the extent of modification required will depend on the granularity of componentization and on how close the new interfaces are to the existing ones.

11 Reusable Scientific Components and Interfaces

Before we can begin to realize our vision of interoperable computational science components, we must address many difficult research issues. We employ a two-pronged approach to scientific component development, namely (1) defining domain-specific interface specifications and (2) developing a suite of parallel scientific components. All this work

is in collaboration with applications scientists and domain specialists, with whom we are also exploring research issues including quality of service issues related to robust, efficient, and scalable performance.

11.1 Common Interface Development

The development of common interfaces that a large number of tools support is critical to the notion of plug-and-play scientific computing. Common interfaces are especially important for scientific data components, as the interfaces for numerical tools and application components often include a certain degree of specificity of the data structures used as input and output arguments. We are working with domain experts in the Terascale Simulation Tools and Technologies (TSTT) [Glimm et al., 2001] and APDEC [Colella, 2003] groups to develop common interfaces for two broad areas of scientific data components, namely interfaces for structured, unstructured, and adaptive mesh access and a descriptor for dense arrays distributed across the processors of a parallel computer. These data-centric components will define the parallel layout of data across processors and form the *lingua franca* for other components that manipulate the data. We are also working in conjunction with the Terascale Optimal PDE Simulations (TOPS) [Keyes, 2004] group to define component interfaces for linear, nonlinear, and optimization solvers. In addition, we engage the high-performance scientific community at large to participate in similar dialogs in their particular areas of expertise.

11.2 High-Performance Components

We have developed (internally and through collaboration) high-performance production components that are used in scientific applications as well as prototype components that aid in teaching CCA concepts. A common theme in this work is combining application-specific components, as introduced in the previous sections, with more general-purpose ones that can be reused across a range of applications. These freely available components include various service capabilities, tools for mesh management, discretization, linear algebra, integration, optimization, parallel data description and redistribution, visualization, and performance evaluation. We have also developed a variety of component-based scientific applications that demonstrate component reusability and composability, including several that employ domain-specific interfaces being defined by CCA working subgroups for scientific data and parallel data redistribution. Several of these applications implement PDE-based models using either adaptive structured meshes or unstructured meshes, while others solve unconstrained minimization problems that arise in computational chemistry.

These components and applications, many of which are available as Linux RPMs via <http://www.cca-forum.org/software.html>, serve as part of CCA tutorial material and provide a starting point for interfaces in several scientific applications. This software has been developed on a strong foundation in the form of rich parallel tools that already used abstractions in their design, including CUMULVS [Kohl and Geist, 1999; Geist et al., 1997], Global Arrays [Nieplocha et al., 1996; Pacific Northwest National Laboratory, 2004a], GrACE [Parashar et al., 2004], CVODES [Hindmarsh and Serban, 2002], MPICH [Argonne National Laboratory, 2003], PETSc [Balay et al., 2003; Balay et al., 1997], PVM [Geist et al., 1994], and TAO [Benson et al., 2001; Benson et al., 2003b]. A partial list of components follows.

11.2.1 Utilities and Services

- *Services in Ccaffeine.* B. Allan, R. Armstrong, S. Lefantzi, and E. Walsh (SNL), M. Govindaraju (Binghamton). The CCA specification treats framework services exactly like CCA components except that the port that embodies the service is *always* connected to the component. A number of such services are available in the Ccaffeine framework. The parameter port service is the most commonly used, and it allows components to have parameters set on them interactively by a user. Another service allows the connection of the original “classic” (C++ only) ports to Babel components, thereby accommodating legacy CCA software. Additional utility services include permitting a component to access MPI, to receive connection events, and to establish its own interactive window with a user.
- *Performance Observation.* S. Shende and A. Malony (University of Oregon), C. Rasmussen and M. Sottile (LANL), and J. Ray (SNL). The TAU (Tuning and Analysis Utilities) performance observation component (version 1.1) provides measurement capabilities to components, thereby aiding in the selection of components and helping to create performance aware intelligent components; see [Shende et al., 2003; Ray et al., 2003b] for further details. This component is currently used in combustion applications discussed in Section 13.1, and future plans include incorporation into a variety of other simulations, including the quantum chemistry application discussed in Section 13.3, to provide comprehensive inter- and intra-component performance instrumentation, measurement and analysis capabilities.

11.2.2 Data Management, Meshing, and Discretization

- *Global Arrays*. M. Krishnan and J. Nieplocha (PNNL). Many scientific applications rely on dense distributed arrays. The Global Array (GA) library [Nieplocha et al., 1996; Pacific Northwest National Laboratory, 2004a] provides an extensive set of operations on multidimensional dense distributed arrays. A rather unique capability is the support in GA for the shared memory programming model, where arrays can be accessed as if they were located in shared memory. This is accomplished based on the one-sided communication operations that transfer data between local memory of the calling processor and the arbitrary sections of distributed/shared arrays. We developed a `GlobalArray` component that provides interfaces to full capabilities of GA, including the data- and task- parallel operations. This component provides three ports: `GlobalArrayPort`, `DADFPort` and `LinearAlgebraPort`. `GlobalArrayPort` offers interfaces for creating and accessing distributed arrays. These interfaces are intended to support the collection of global information and creation of `GlobalArray` objects. All details of the data distribution, addressing, and data access are encapsulated in the `GlobalArray` objects. The `LinearAlgebraPort` provides core linear algebra support for manipulating vectors, matrices, and linear solvers. Some of the linear algebra operations are implemented internally, and others are provided through interfaces to third-party parallel linear algebra libraries such as ScaLAPACK [Blackford et al., 1997]. `DADFPort` offers interfaces for defining and querying array distribution templates and distributed array descriptors following the API proposed by the CCA Scientific Data Components Working Group. The `GlobalArray` component is currently used in applications involving molecular dynamics and quantum chemistry, as discussed in Sections 6 and 13.3; further details are in [Benson et al., 2003a].
- *TSTTMeshQuery*. L. F. Diachin (LLNL). This component is a prototype of an unstructured, triangular mesh component that supports the TSTT mesh query interface [Glimm et al., 2001] for access to node and element geometry and topology information; opaque tags support user-defined data; further information is in [Norris et al., 2002]. This interface is sufficient to implement linear, finite-element discretization for diffusion PDE operators. This component will be expanded to several TSTT-compliant mesh components built from existing DOE software that support a wide range of two and three-dimensional meshes. Such components will be used to demonstrate the utility of interchangeable and interoperable meshing infrastructures in the solution of PDE-based applications.

- *FEMDiscretization*. L. F. Diachin (LLNL). This component provides linear, finite-element discretizations for commonly used PDE operators and boundary conditions. It currently employs unstructured triangular meshes through the `TSTTMeshQuery` component and provides matrix and vector assembly routines to create linear systems of equations in simple PDE-based applications; see [Norris et al., 2002] for details. The `FEMDiscretization` component approximates advection and diffusion operators as well as Dirichlet and Neumann boundary conditions with either exact or Gaussian quadrature. This component uses the `TSTTMeshQuery` and `LinearSolver` ports. This interface is expected to evolve as the TSTT discretization library is developed, and this prototype component will be replaced with a more sophisticated variant that supports multiple discretization schemes and mesh types. Such components will be used to demonstrate the utility of interchangeable and interoperable discretization strategies in the solution of PDE-based applications.
- *GrACEComponent*. J. Ray (SNL). This component discretizes a domain with a SAMR mesh and implements regriding to preserve resolution and load-balance the mesh. As a wrapper around the GrACE library [Parashar et al., 2004] developed by M. Parashar of Rutgers University, `GrACEComponent` takes care of all the geometric aspects of the mesh (size and location of patches, their resizing due to regriding, and their placement on processes for load-balancing). This component also serves as a factory for a “Data Object” that contains data on all the patches. The data object takes care of message passing for ghost cell updates. This component is used in the combustion applications discussed in Section 13.1, and will continue to evolve to incorporate the latest GrACE features.
- *HODiffusion and SpatialInterpolations*. C. Kennedy and J. Ray (SNL). These components use an underlying Fortran 77 library developed by C. Kennedy that implements higher-order (orders 2 – 8) finite difference stencils, including both first and second derivatives. `HODiffusion` supports both symmetric and skewed stencils for collocated and staggered output and calculates higher order diffusion fluxes using these stencils. The `SpatialInterpolations` component supplies the prolongation and restriction operators between SAMR patches at two adjacent levels of refinement, where the order of interpolation has to be commensurate with the spatial discretization in `HODiffusion`. These components currently handle diffusion transport subassembly in the combustion applications discussed in Section 13.1; future plans include continued testing on hierarchical grids.

11.2.3 Integration, Optimization, and Linear Algebra

- *CvodesComponent*. R. Serban (LLNL). This component includes ports both for a generic implicit ODE integrator (`OdeSolverPort`) and for an implicit ODE integrator with sensitivity capabilities (`OdeSolverSPort`). `CvodesComponent` is based on CVODES [Hindmarsh and Serban, 2002] and is used for chemistry integration in the combustion applications discussed in Section 13.1.
- *TAOSolver*. S. Benson, L. C. McInnes, B. Norris, and J. Sarich (ANL). This component implements a simple `OptimizationSolver` interface for unconstrained and bound constrained optimization problems; see [Norris et al., 2002] and [Benson et al., 2003a] for details. The underlying optimization solvers are provided by the Toolkit for Advanced Optimization [Benson et al., 2001; Benson et al., 2003b] and include Newton-based methods as well as limited-memory variable-metric algorithms that require only an objective value and first order derivative information. `TaoSolver` employs external components for parallel linear algebra, where current support includes both Global Arrays and PETSc. `TaoSolver` is used within applications involving molecular geometry optimization and molecular dynamics, which are further discussed in Sections 6 and 13.3. `TaoSolver` is the basis for an evolving optimization solver component that will employ linear algebra interfaces under development within the TOPS group.
- *LinearSolver*. B. Norris (ANL). This component provides a prototype port for the solution of linear systems. These interfaces are in the process of evolving to support common interfaces for linear algebra that are under development within the TOPS group. Future work will include transitioning this component, as well as others such as `TaoSolver` and `FEMDiscretization`, to use the new TOPS interfaces, so that they can easily benefit from the full suite of linear algebra software available within the TOPS center.

11.2.4 Parallel Data Description, Redistribution, and Visualization

- *DistArrayDescriptorFactory*. D. Bernholdt and W. Elwasif (ORNL). This component provides a uniform means for applications to describe dense multi-dimensional arrays and is based upon emerging interfaces from the CCA Scientific Data Components Working Group.
- *CumulvsMxN*. J. Kohl, D. Bernholdt and T. Wilde (ORNL). This component builds on CUMULVS [Kohl and

Geist, 1999; Geist et al., 1997] technology to provide an initial implementation of the $M \times N$ parallel data redistribution interfaces that are under development by the CCA $M \times N$ Working Group. `CumulvsMxN` is designed to span multiple CCA frameworks and to pass data between two distinct parallel component applications. See Section 12 for further information.

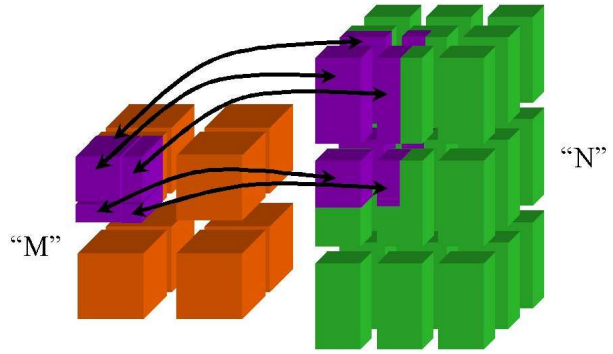
- *ParticleCollectionFactory*. J. Ray (SNL) and J. Kohl (ORNL). This component is a prototype for doing $M \times N$ parallel data redistribution on combustion data for use in post-processing in the combustion applications discussed in Section 13.1. The `ParticleCollectionFactory` component imitates a patch on a SAMR grid as a “particle,” which can then be employed for data redistribution and post-processing. Future plans include increasing the robustness of the code and using it in off-machine, concurrent post-processing.
- *VizProxy* J. Kohl and T. Wilde (ORNL). This component provides a companion $M \times N$ endpoint for extracting parallel data from component-based applications and then passing this data to an external front-end viewer for interactive graphical rendering and exploration. Variants provide general-purpose components for interactive visualization of data based on structured meshes as well as unstructured triangular meshes; support for particle-based data is under development. Using the CUMULVS viewer library and protocols, a variety of commercial and public domain visualization tools can be utilized at the front-end user interface [Wilde et al., 2002; Wilde et al., 2003]. Currently provided front-ends include a simple 2D “slicer” viewer and a 3D viewer for AVS 5; additional viewers are under development for VTK, AVS/Express and the CAVE.

12 $M \times N$ Parallel Data Redistribution

As parallel simulations are increasingly coupled together to form complex multi-physics and multi-scale applications, developers face a growing need to be able to efficiently manage the movement of distributed data from one simulation component to another. Typically, the sending and receiving sides in such coupling utilize different distributions of data across their parallel environments, and when such coupled simulations are run in a multiple program/multiple data paradigm, even the number of parallel processors on each side may well be different (say M and N processes, respectively). The CCA refers to this as the “ $M \times N$ parallel data redistribution” problem, illustrated in Figure 6.

Solving the $M \times N$ problem is a complex and tedious programming task, which involves determining the mapping

Figure 6: A schematic of an $M \times N$ parallel data redistribution problem. In this case, $M=8$ and $N=27$, and the dark patches illustrate the fact that data originating on a single processor may be redistributed across many processors on the receiving end.



of data between the source and destination data decompositions (often referred to as the communication schedule) and then efficiently implementing a custom all-to-all communication using this schedule. At present, the $M \times N$ coupling in most such applications is done in an application-, or at least domain-specific fashion, limiting the type of data distributions supported and other aspects of the problem. Often, separate $M \times N$ couplers are created for each pair of models coupled in a multi-physics simulation, with no generalized or reusable solutions.

In so far as a component-based environment is intended to facilitate, among other things, the development of large coupled simulations, the development of a more general approach to the $M \times N$ problem is an important research area within the CCA effort. Our long-range view of this research bridges applications, components, interfaces, and the core CCA environment. We are starting with the development of interfaces and component-based implementations that provide a generalized $M \times N$ capability, which applications can call to affect the transfer of data from one group of components to another. Another approach involves integrating the mechanics of this data redistribution directly into the CCA framework, thereby removing from the user the need to explicitly invoke the data transfers. This approach can also be generalized into “parallel remote method invocation” (PRMI), in which not only are data redistributed among disparate process groups, but method invocations themselves are coordinated between distinct groups of parallel processes. Finally, we also look beyond “simple” data redistribution toward ways of generalizing the additional processing that often takes place at the interfaces between coupled simulations, such as mesh interpolation, time or spatial averaging, unit conversion, and various other filtering operations. Here we provide an overview of the current status of progress toward general component- and framework-based $M \times N$ capabilities.

12.1 $M \times N$ Interfaces and Component Implementations

The CCA Forum's $M \times N$ Working Group has developed a prototype interface specification for an $M \times N$ component. This effort has drawn heavily on experiences with two particular tools that offer fairly general $M \times N$ -like capabilities, based on libraries rather than components: CUMULVS [Kohl and Geist, 1999; Geist et al., 1997; Kohl, 1997; Kohl and Papadopoulos, 1995] and PAWS [Keahey et al., 2001; Beckman et al., 1998]. These tools utilize complementary models of parallel data sharing and coupling. PAWS is built on a “point-to-point” model of parallel data coupling, with matching “send” and “receive” methods on corresponding sides of a data connection. CUMULVS is designed for interactive visualization and computational steering, and so provides protocols for persistent parallel data channels with periodic transfers, using a variety of synchronization options.

The current $M \times N$ interface therefore allows the specification of both “one-shot” and “persistent” periodic data transfers and is structured to allow for “third-party” control of a data redistribution connection. This possibility minimizes the modifications necessary for the sending and receiving components to make them “ $M \times N$ aware.” The minimal changes involve only providing a descriptor of their parallel data distribution for each data object of interest, and then “instrumenting” the simulation code with calls to the $M \times N$ component's `dataReady()` method at points where the data is in a “consistent” state globally. Such points in the computation occur where it would be appropriate to transfer data to the recipient, or at points where it would be appropriate for the recipient to accept new data, respectively. Control over the details of the connection – how often data exchange occurs, or even whether it occurs at all, can all be placed into a third component if desired.

Implementations of the prototype $M \times N$ interface have been built on top of both CUMULVS and PAWS, and they have been used in a number of models and actual scientific simulations. This experience, together with that from significant application- or domain-specific $M \times N$ implementations (see for example the Distributed Data Broker and Model Coupling Toolkit discussed in Section 13.2) and other related efforts (i.e., Chaos [Lee and Sussman, 2004; Edjlali et al., 1997; Ranganathan et al., 1996]) are now being assessed as part of an effort to refine the interface specification and create more general and more efficient implementations.

12.2 Framework-Based $M \times N$

Although the component-based $M \times N$ capability is powerful, it can also be somewhat cumbersome to use, since it requires users to modify their code and explicitly manage the transfers. A much more user-friendly approach, which is also being actively pursued in the CCA, is to subsume the $M \times N$ data transfer capabilities *into* the CCA framework (i.e., as an optional CCA service) and make the framework responsible for ensuring that data is appropriately redistributed when method calls are made that require it.

Framework-based $M \times N$ possibilities raise a number of issues. One is the fact that the framework must somehow obtain data distribution information for the sending and receiving data objects. A logical approach, which has been used in the past (see [Keahey and Gannon, 1997]), is to introduce such information into the interface definition. Research using Utah's SCIRun2 CCA framework extends the SIDL language with primitives for parallel data redistributions and also parallel remote method invocation. A modified version of Babel parses the extended SIDL specifications and generates appropriate glue code to handle the data redistribution when method invocations are made [Damevski, 2003].

A second issue, alluded to above, is the fact that the framework-based approach extends the problem from "simple" data redistribution to parallel remote method invocation. An extremely challenging and complex research task is both the specification and implementation of an environment in which arbitrary subsets of M processes can, in parallel, invoke methods on arbitrary subsets of N processes, including the ability to designate the appropriate redistribution for all of the method arguments. In addition to the aforementioned work using SCIRun2, with PRMI directives in the SIDL, researchers at Indiana University have developed DCA [Bertrand and Bramley, 2004], an experimental distributed CCA framework based on MPI, as a tool to explore PRMI in greater detail.

13 Scientific Applications

The groups adopting the Common Component Architecture environment for the development of their applications do so for a variety of reasons. For example, they may be particularly interested in the black-box nature of components to facilitate the creation and evolution of a computational toolkit. They may need more efficient and flexible ways to assemble complex coupled simulations or want to refactor code to make it more modular. They may be interested in

increasing interoperability with other similar applications in the same scientific domain. Alternatively, they may want to more easily leverage software developed by experts in other domains.

In this section, we present overviews of CCA-based application development activities in the areas of combustion simulation, global climate modeling, and quantum chemistry. These simulations illustrate the diversity of CCA applications and reasons for adopting CBSE for scientific computing.

13.1 Combustion Modeling

One of the most successful implementations of the CCA paradigm to date is in combustion modeling. The endeavor, which started in 2001 within the context of the SciDAC-funded [U. S. Dept. of Energy, 2003] Computational Facility for Reacting Flow Science (CFRFS) project [Najm et al., 2003], seeks to create a facility for the high fidelity simulation of flames, involving realistic physical models, nonlinear PDEs, and a spectrum of time and length scales. Given the complexity of the problem and the multiplicity of physical and mathematical models required for the task, a component based approach was clearly indicated. CCA was chosen primarily for its high performance and simplicity.

The target is the modeling of combustion problems defined in simple geometries, i.e., logically rectangular domains, employing structured adaptively refined meshes (SAMR). Evolution in time of the flow quantities (e.g., density, momentum, temperature, species concentrations, etc.) is governed by a set of nonlinear PDEs, which have the general form:

$$\frac{\partial \Phi}{\partial t} = \mathbf{F}(\Phi, \nabla \Phi, \nabla^2 \Phi, \dots) + \mathbf{G}(\Phi), \quad (1)$$

where Φ is the vector of flow variables at a given mesh point. Numerically, \mathbf{G} involves the variables only at the same mesh point, while \mathbf{F} involves spatial derivatives which are computed using finite difference or finite volume schemes and consequently depend upon the mesh point and its close neighbors. The dependence of \mathbf{F} on neighboring points requires that steep variations of Φ be fully resolved in scattered time-evolving regions of the domain, which is usually achieved by increasing the grid density locally. The SAMR technique that we employ overlays a uniform coarse mesh on the domain and dynamically generates a mesh hierarchy of rectangular patches with different grid densities, based on an error threshold suitably defined to capture the errors caused by the coarseness of the mesh during the computation. Physically, for the reacting flow systems described by Equation 1, \mathbf{F} expresses diffusion and convection in the flow, and \mathbf{G} expresses the chemically reacting source terms of the flow.

In order to design a component-based software infrastructure to model reacting flow systems, one needs to consider and classify two aspects of the simulation: the physics of the system and the numerical scheme that will be employed. This classification determines the software subsystems of the infrastructure, that is, collections of components that embody specific physical or numerical functionality. For the class of reacting flow problems in which we are interested, the main physics sub-problems are: diffusion and convection (**F**), and chemical reactions (**G**). Numerically **G** (Equation 1) is stiff, i.e. the ratio of the largest and the smallest eigenvalues of $\partial \mathbf{G} / \partial \Phi$ is large, while **F** is non-stiff. We address the time integration in a “decoupled” manner by employing an operator-splitting scheme [Strang, 1968; Sportisse, 2000]. We solve the stiff problem (**G**) using an implicit backward difference formulation and the non-stiff problem (**F**) using an explicit integrator. Thus, the numerical scheme naturally divides into two separate integration software subsystems: the *Implicit Integration Subsystem* and the *Explicit Integration Subsystem* (Fig. 7a).

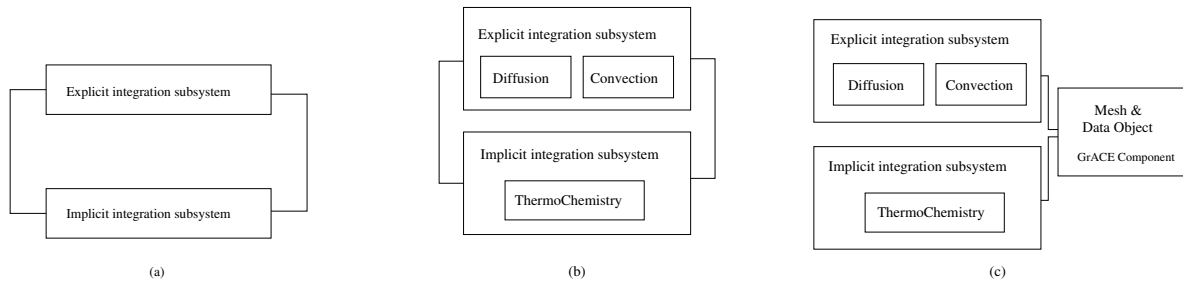
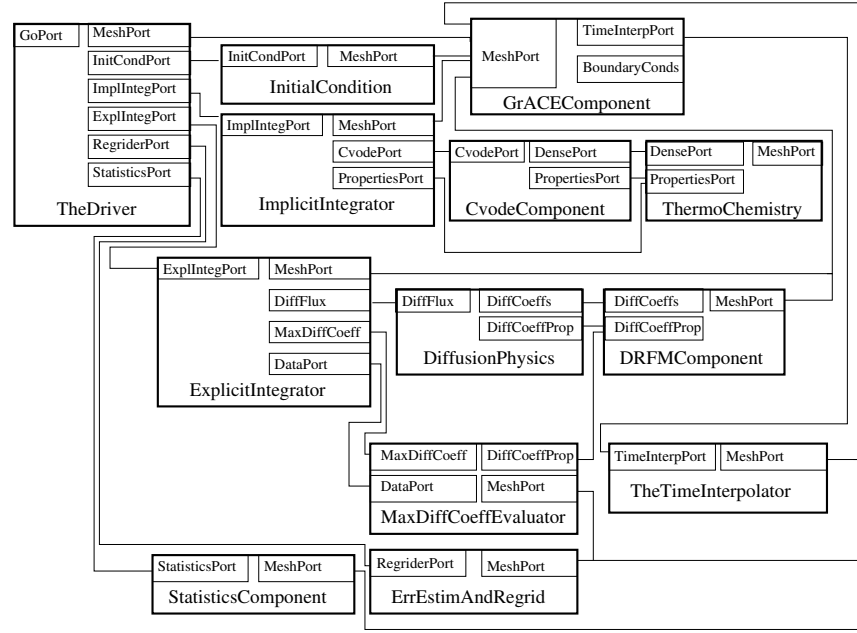


Figure 7: Decomposition sequence for the combustion modeling software. (a) shows the coarse decomposition along numerical lines into explicit and implicit integration subsystems. In (b), the dictates of physics result in the separation of physical processes as separate subsystems, which are then populated by various elementary physical and mathematical models. In (c), the domain decomposition and domain discretization functionalities are formally removed to form new subsystems. The process follows closely the general approach described in Section 10.1.

The physics of the problem, on the other hand, dictate a finer decomposition: for the Explicit Integration Subsystem a *Diffusion Subsystem* and a *Convection Subsystem*, and for the Implicit Integration Subsystem a *Chemical Reactions Subsystem* (Figure 7b). Also, the code is fully parallel with structured adaptive mesh refinement; we need to employ a “data object” to account for efficient domain decomposition and a “mesh object” to perform the adaptive mesh regeneration. Both objects are accommodated by the componentized version of the GrACE library [Parashar et al., 2004] as a separate software subsystem (Figure 7c). This path of classification results in the first design approximation for the reacting flow toolkit. The next step is a further decomposition of the software subsystems into components, resulting in a complex software infrastructure where each individual module is reusable and easy to maintain. Details

are beyond the scope of this paper and can be found in [Lefantzi et al., 2003b]. A schematic reaction-diffusion code assembly is presented in Figure 8.

Figure 8: The final results of the decomposition described in Figure 7 for a reaction-diffusion problem. Large boxes represent components, smaller rectangles represent *provides* ports (left justified within component box) and *uses* ports (right justified). Lines connect *uses* and *provides* ports. Details can be found in [Lefantzi et al., 2003b].



Our experience with developing CCA-based scientific components for combustion modeling has demonstrated the following:

1. General purpose components, implementing a particular numerical or physical functionality, are reused in various code assemblies. References [Lefantzi et al., 2003b; Lefantzi and Ray, 2003] document our experience with numerical integration components (implementing a number of explicit Runge-Kutta and backward difference algorithms), physical models (diffusion coefficients of gaseous mixtures calculated via a number of different approaches), data and mesh objects, along with the rationale for particular code assemblies.
2. Once a particular decomposition has materialized, the modular nature of component-based codes allows the easy replacement of an individual component by another that supplies the same functionality but may exhibit different characteristics (such as higher accuracy). References [Lefantzi et al., 2003b; Ray et al., 2003a; Lefantzi et al., 2003a] document how we exploited this modular nature to experiment with multiple models of diffusion coefficients of gaseous mixtures (using constant Lewis number approximations, mixture averaged models based on Lennard-Jones potentials, etc.) and higher order spatial discretizations in the evaluation of the diffusion term (\mathbf{F} in Equation 1).

3. One of the most significant advantages offered by components is the ability to perform “unit” testing, i.e., each component is tested individually, in the absence of the final assembly where they will be typically used. This simple testing process with its quick turn-around time allows one to debug and incorporate a verified component in the final assembly easily. While this approach is not particular to components, their modular nature allows their incorporation into the final assembly *without any code modification*, thus drastically reducing the possibility of introducing errors.

13.2 Global Climate Modeling

Simulation using coupled climate models is the chief tool used by scientists to understand the Earth’s climate system’s past (i.e., paleoclimate), its internal variability and sensitivity to external forcings such as insolation and trace chemical concentrations (e.g., carbon dioxide), and to construct scenarios for future climate change. The fact that climate scenarios are constructed from large history datasets resulting from long model integrations that span the century-to-millennial timescale makes climate modeling a grand challenge computational science problem. The high degree of complexity in the number of physical processes modeled and the numerous interactions among the system’s components—atmosphere, ocean, sea-ice, river, biosphere, and cryosphere—result in a correspondingly high level of software complexity. For these reasons, the climate community is examining methods to better write and maintain their applications. Several of these efforts employ the CCA in various aspects of the problem [Larson et al., 2004b].

13.2.1 UCLA Coupled Climate Model and Distributed Data Broker

An ongoing project at JPL consists of a demonstration of CCA with three specific climate components: the UCLA Atmospheric General Circulation Model (AGCM) [Wehner et al., 1995], the LANL POP Ocean General Circulation Model (OGCM) [Smith et al., 1992], and the UC Berkeley/UCLA Distributed Data Broker (DDB) [Drummond et al., 2001]. Neither the AGCM nor OGCM will be described in detail in this paper. In brief, they are both explicit time-marching schemes that read in initial conditions, have boundary conditions that are defined as fixed except at the ocean-atmosphere interface (sea surface), where they need to exchange data, and then start marching forward in time. The atmosphere and ocean exchange flux and state data at the ocean-atmosphere interface. Flux data exchanged include radiative, fresh water, and momentum fluxes. State data include sea surface temperature and ocean albedo.

The DDB is not as well known as the AGCM and OGCM, so it is described in a bit more detail. The DDB is used in two phases. First, components that will use the DDB to exchange data must register with it by signing up to produce and/or consume data. In this phase, they also describe the global view of the data that they have/want, and the mapping of this data onto the processors. The DDB then calculates what messages will have to be sent to transfer data from components that produce it to components that consume it. This calculation is done on a single processor at the end of this first phase, but the information about the messages themselves is stored on the processors that will be sending or receiving data, and the actual data exchange is also distributed. In the second phase, the actual messages are sent when the components on a given processor signal that they have produced or are ready to consume data. Also, any interpolation (needed if the grids are not coincident or if no data exists at some set of points) is done by the DDB.

The usual approach to running a simulation using this coupled climate model would be as a multiple program/multiple data (MPMD) application, with the AGCM and OGCM each running on subsets of the processors, and the DDB running on all processors to couple the AGCM and OGCM. In this case, the approach chosen to implement the componentized version using the Ccaffeine framework [Allan et al., 2002] is to cast the AGCM as the overall driver of the simulation (providing a `Go` port and using ports provided by the DDB), and to have the AGCM determine the processes on which the OGCM should be run. An alternative approach would be to move the driver functionality and partitioning decisions from the AGCM to a separate driver component and use a full MCMD model for the application. Demonstrations of such capabilities have been developed for the Ccaffeine framework at the same time as this work was being carried out and have not yet been incorporated.

The DDB component is a domain-specific solution to what the CCA refers to as the “ $M \times N$ parallel data redistribution” problem. In addition to being directly useful in other applications, both in climate and other fields, experience with this component also informs the CCA’s efforts to develop a more general parallel redistribution capability (Section 12).

13.2.2 The Earth System Modeling Framework

The Earth System Modeling Framework (ESMF) [Killeen et al., 2003] is a national effort to develop common software utilities (“ESMF infrastructure”) and coupling services (“ESMF superstructure”) in the climate, weather, and data assimilation domains. The fifteen application testbeds for the ESMF include the Community Climate System Model

(CCSM), the Weather Research and Forecast (WRF) model, models from the NOAA Geophysical Fluid Dynamics Laboratory, the National Centers for Environmental Prediction, MIT, and the NASA Global Modeling and Assimilation Office. Each of these applications either is a multi-component, coupled modeling system or has the potential to be used in such a modeling system. As collaborators on the ESMF project, these groups will use the ESMF component constructs to represent their atmosphere, ocean, land, sea-ice, data assimilation, and coupler components and the ESMF regridding and communication utilities to connect these components into applications.

The objectives of ESMF are to facilitate:

- an extensible, hierarchical architecture for structuring complex climate, weather and data assimilation models;
- interoperability of modeling components among major centers;
- lower cost of entry into Earth system modeling for university and smaller groups; and
- sharing of common software utilities and services.

Next-generation Earth modeling systems require the incorporation and evaluation of new physical processes such as biogeochemistry and atmospheric chemistry, which may be represented as new sets of software components. Thus, the advances that the ESMF offers – the ability to organize large models, easily incorporate new components, and compare different component implementations (for example, different representations of the ocean) – are critical capabilities for ongoing scientific progress.

Like the CCA, the ESMF provides a generic component abstraction that connects user-defined components to the framework through a `SetServices()` method. Unlike the CCA, the ESMF customizes methods and data structures for Earth system models, and it provides utilities and services specific to the climate/weather/data assimilation domain. For example, the data exchanged among ESMF components is packed into a data structure called an `ESMF_State`, which is a container for data types that represent geophysical fields. Since ESMF applications typically perform a setup, time step through numerical solution of PDEs, and then shut down, ESMF codes are required to have `Initialize()`, `Run()`, and `Finalize()` methods.

The ESMF and CCA groups have worked closely together to ensure interoperability between the ESMF and CCA frameworks; i.e., to guarantee that ESMF components may be used within the CCA, and that the ESMF will be able to utilize the wide variety of CCA-compliant components that exist or are under development. An ESMF-CCA prototype,

described in more detail later in this section, has been developed that uses the component registration methodology and GUI from the CCA. ESMF components do not need to be modified, but can simply be wrapped to become CCA-compatible. This prototype demonstrates that interoperability between the CCA and the ESMF is entirely plausible and in fact not onerous, as similar architectural principles underlie both frameworks. The capacity for interoperability between these frameworks is exciting because it opens the possibilities of bringing many Earth system modeling components into the CCA arena, and of making CCA tools available to a host of Earth system applications.

ESMF-CCA Interoperability Several notable distinctions exist between the CCA and the ESMF component frameworks. First and foremost, while the CCA provides a general component model, the ESMF provides a specialized component model tailored to a specific application domain. Additionally, the CCA enables dynamic composition of component-based applications at run time, while the ESMF dictates static linkage of a component hierarchy. On the other hand, the CCA, by its nature, does not provide concrete classes with which to build components, while the ESMF does (the ESMF infrastructure). In this sense, the CCA is component framework, while the ESMF is an application framework.

These differences suggest the intriguing possibility of using the CCA component model and the ESMF application framework together to build climate-related applications, thereby bringing the dynamism of the CCA component model to this community. This cooperation would also enable coupling to CCA components from within the ESMF framework, allowing ESMF components to step outside of bounds of the `ESMF_Initialize()`, `ESMF_Run()`, `ESMF_Finalize()` paradigm if a richer component interface is required. Furthermore, the union of these two frameworks would provide the CCA community with access to ESMF components.

Bridging the CCA and ESMF to create a joint ESMF-CCA component model is fairly straightforward. ESMF components are wrapped with a thin layer of C++ code to provide the CCA component veneer. This wrapper allows an ESMF-CCA application to be composed by selecting from a palette of ESMF-CCA components using the Ccaffeine CCA framework. A special version of the ESMF component registration function is provided for all ESMF-CCA components, to allow them to register their ESMF interface functions (initialize, run, finalize). Once all components have been created and connected, the CCA framework passes control flow over to the ESMF framework.

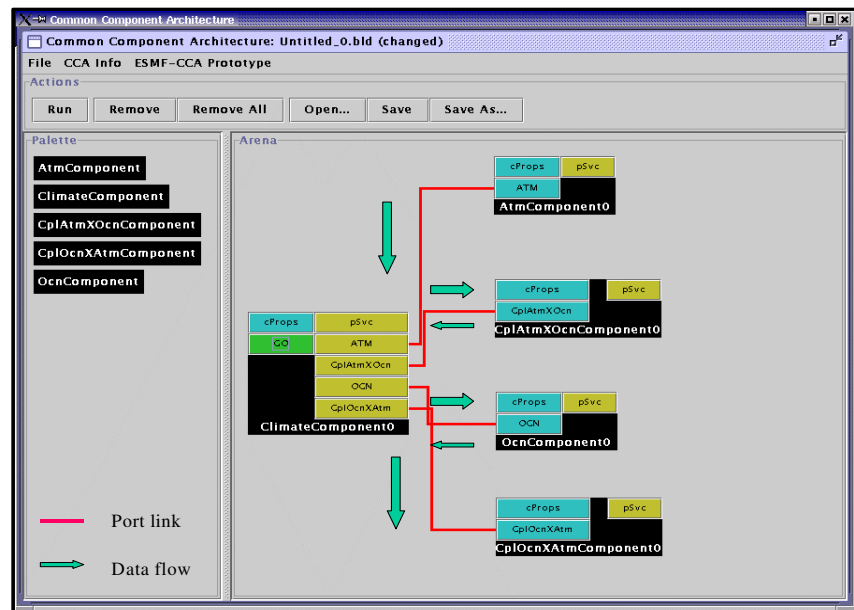
A major advantage of this dynamic approach is that components can be easily substituted for one another without modifying user code (as long as the swapped components conform to the same interface). As it now stands, one must

modify ESMF superstructure components to replace a component, as the complete component hierarchy is coded into an ESMF application. We recognize that climate and weather models do not typically require run-time swapping of geophysical components, since it takes on the order of months to tune and validate these applications, and months to run long climate simulations. However, run-time swapping may be useful for communication and input/output components.

ESMF Prototypes Using the CCA To demonstrate possibilities for interoperability between the ESMF and the CCA, we have developed a prototype ESMF implementation on top of CCA tools and CCA-compliant components [Zhou et al., 2003; Zhou, 2003]. The CCA components additionally provide the basic methods required by the ESMF (initialize, run, finalize). Data exchange occurs through a self-describing data type similar to the ESMF's `ESMF_State`. We are currently working on a C++ version of the ESMF component interface so that ESMF components can be used with the CCA in a straightforward way. A model ESMF-CCA application has been implemented using the Ccaffeine framework; this project illustrates the case of sequential coupling of an atmosphere model and an ocean model (Figure 9). At the beginning of a simulation, five components are created: a driver (Climate), the atmosphere (Atm), the ocean (Ocn), a coupler from atmosphere to ocean (CplAtmXOcn), and a coupler from ocean to atmosphere (CplOcnXAtm). The component execution sequence of events is as follows:

1. The atmosphere component provides its data at its boundary, `exportAtm`, to the coupler, `CplAtmXOcn`.
2. `CplAtmXOcn` uses `exportAtm`, transforms it into `importOcn` with interpolation routines, and then provides `importOcn` to the ocean component.
3. With `importOcn`, the ocean component runs its solver for the evolution equations and then provides its data at the boundary, `exportOcn`, to the coupler, `CplOcnXAtm`.
4. The coupler, `CplOcnXAtm`, uses `exportOcn`, transforms it into `importAtm`, and provides `importAtm` to the atmosphere component.
5. The atmosphere component uses `importAtm`, runs its solver for evolution equations, and then provides its data at the boundary, `exportAtm`, to the coupler, `CplAtmXOcn`.

Figure 9: CCA wiring diagram showing component relationship in a simulation of coupled atmosphere and ocean model components of the ESMF-CCA prototype.



13.2.3 The Community Climate System Model

The Community Climate System Model (CCSM) [Harper and Kauffman, 2004] is a coupled climate model, comprising several mutually interacting models developed through an interagency collaboration between the U.S. Department of Energy and the National Science Foundation's National Center for Atmospheric Research. Key modules of the CCSM include models for atmosphere, ocean, sea ice, land-surface, river routing, and a flux coupler. The flux coupler provides the overall coordination of the coupled model's execution and a variety of services related to the simulation data: intermodel state and flux data transfer, interpolation between model grids, calculation of fluxes and other diagnostic quantities, time averaging and accumulation of flux and state data, and merging of flux and state data from multiple model sources for use by another model.

Collaboration between the CCSM and the CCA is focused on three areas. We are prototyping use of CCA at the system integration level, the model subcomponent level, and the algorithmic level. We are also exploring the use of CCA to package portions of the Model Coupling Toolkit [Larson et al., 2004a; Larson et al., 2001; Ong et al., 2002] (the foundation code on which CCSM's flux coupler is built) as CCA components. Since CCSM plans to adopt the standard interfaces being developed by the ESMF project (see Section 13.2.2), this work is also being performed in collaboration with the ESMF group. The next three sections describe individual projects in this area.

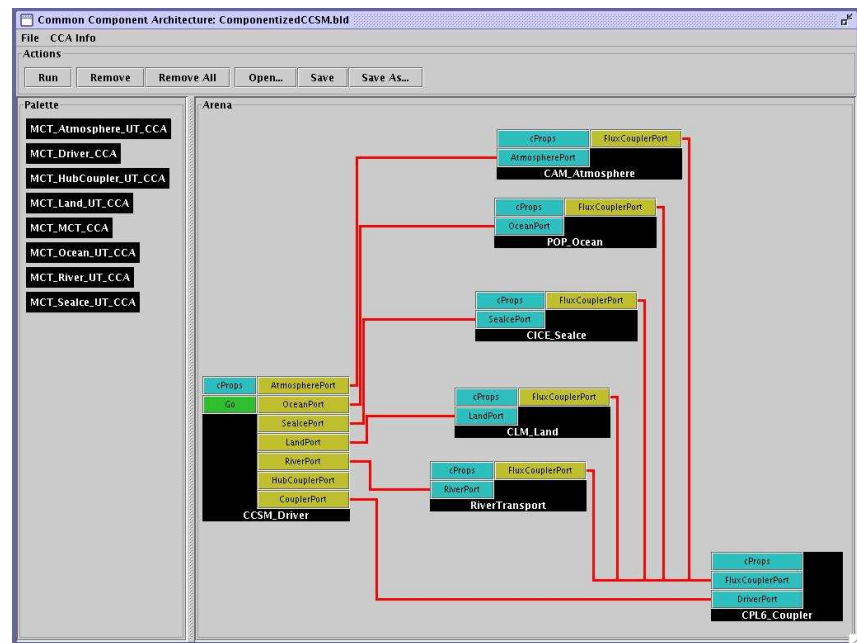
Model Coupling Toolkit The current CCSM flux coupler was implemented using the Model Coupling Toolkit (MCT) [Larson et al., 2004a; Larson et al., 2001; Ong et al., 2002]. MCT is a software package for constructing parallel couplings between MPI-based distributed-memory parallel applications which supports both sequential and concurrent couplings and can support multiple executable images if the implementation of `mpirun` used supports this feature. MCT is implemented in Fortran 90, and its programming model is friendly to scientific programmers, based on a Fortran 90 module to declare MCT-type variables and the invocation of MCT routines to create couplings.

The first major focus of the MCT-CCA collaboration has been the creation of a migration path between MCT-based coupling and a CCA-compliant component-based approach [Larson et al., 2004c]. The approach we are taking involves the use of the ESMF specification to bridge between the MCT and the CCA's more generic component environment. Low-level MCT functionality is being wrapped to provide an implementation of the relevant ESMF-defined interfaces and adapters to ESMF-defined data objects, such as `ESMF_State`. Currently, we support SCMD component scheduling and coupling only, but it will soon be expanded to support MCMD coupling as well.

We have begun the task of describing ESMF-compliant MCT interfaces using SIDL. The first and obvious motivating factor is the goal of repackaging MCT as a collection of CCA-compliant components. Prospective MCT-based components include distributed multi-field data storage, parallel sparse matrix-vector interpolation, global integrals, and averages. In addition to their use in the flux coupler context, such components can be used *within* individual models as well, as described below. Another application of the SIDL interface description is the extension of the MCT's programming model to languages other than Fortran 90. This is an example of how the Babel interlanguage interoperability tool can be used in a context outside of the CCA. Finally, like the Distributed Data Broker, the MCT is clearly another domain-specific solution to the $M \times N$ parallel data redistribution problem for the climate modeling area. The experience gained from the development and use of the MCT, as well as comparisons with the DDB, will inform the CCA's efforts to develop general $M \times N$ capabilities. In addition, once the general $M \times N$ interface is more mature, we anticipate using elements of the MCT as the basis for an implementation of a more general data redistribution component.

CCSM System Integration At the system integration level, we are prototyping the use of the CCA to cast the CCSM's component models as CCA-compliant components. We have developed skeleton components for the atmosphere, ocean, sea ice, land surface, river routing, and flux coupler. These components can then be instantiated and

Figure 10: Prototype of a CCA component-based system integration of the CCSM.



connected by using the Caffeine GUI (Figure 10); their subsequent execution is similar to the ESMF-CCA prototype described above. These skeleton components are being expanded to have the capabilities of the CCSM data models that are used to perform system integration tests on the CCSM coupler. The components will then entail sufficient computational complexity that we can begin to assess in the coupling context any performance costs associated with CCA components versus the current coupler.

Community Atmosphere Model An example of a CCA application targeted at the model subcomponent level is the refactoring of the CCSM's atmosphere model, called the Community Atmosphere Model (CAM) [University Corporation for Atmospheric Research, 2004]. CAM has been refactored to disentangle its dynamical core from its subgridscale physics parameterizations. This split has allowed a proliferation of dynamical cores, and CAM currently has three: a pseudo-spectral method, the Rasch-Williamson semi-Lagrangian method, and the finite-volume Lin-Rood scheme. Standardization of the interfaces to these dynamical cores and the overall subgridscale physics package is under way. Once complete, these software modules can be packaged as CCA components. This will allow one to use the CCA to compose and invoke a stand-alone version of the CAM in which the user may plug in the dynamical core of choice. Furthermore, the developer of a new dynamical core will need only code it to the interface standard, and it will be easily integrated into the model for validation.

River Transport Model The river transport model (RTM) in the CCSM version 2 computes river flow on a 0.5° latitude-longitude grid and uses no parallelism in its calculations. Future science goals for the RTM include support for significantly higher-resolution (up to $100\times$ the number of catchments) and catchment-based unstructured grids, support for transport of dissolved chemical tracers, and inclusion of the effects of water storage in the form of reservoirs. At the current resolution, the lack of parallelism in the RTM calculation does not retard the overall execution of CCSM, but the process of coupling to the RTM, which requires gathering (scattering) data to (from) a single processor when sending (receiving) data to (from) the RTM, does impose a bottleneck. This coupling communications cost exceeds the computation cost of the RTM itself.

To incorporate distributed-memory parallelism in the RTM and meet its science goals, we are developing a completely new implementation of the RTM using the MCT. In this approach, we view the problem of river transport as a directed graph with the nodes representing catchments and the directed edges representing the dominant inter-catchment flow paths. The river transport calculation is then implemented as a parallel sparse matrix-vector multiply, and the load balance for this operation can be computed by using graph partitioning. Mass conservation is ensured through the calculation of global mass integrals before and after the matrix-vector multiply. This problem is easily solved using the MCT. The ESMF-compliant MCT multifield data storage object is used to store runoff and tracer concentration data; the MCT's domain decomposition descriptors are used to describe the load balance; MCT's parallel sparse matrix-vector multiply is used to perform the water and chemical tracer transport; and the MCT's paired global spatial integral facility is used to carry out the mass balance and mass conservation calculations.

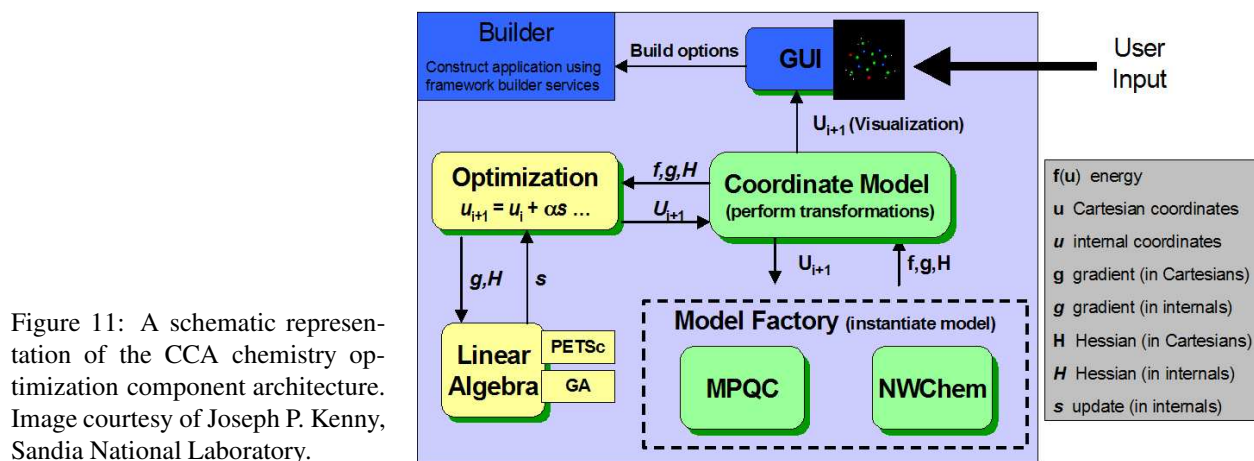
The component-based approach used for key MCT capabilities makes the path to a component-based RTM clear. Four MCT-based components will be used to implement the RTM: a distributed multifield array to store water and chemical tracer concentrations; a parallel sparse-matrix vector multiply to compute water and chemical tracer transport; a component describing the physical grid (i.e., the locations and sizes of the catchments); and a paired spatial integral component to enforce mass conservation.

13.3 Quantum Chemistry

The quantum chemistry community has an extended history of developing large, monolithic codes where essentially all of the code, including the mathematical routines, are written by chemists. This attitude, however, has been changing

– it is becoming much more difficult for any group to maintain expertise in all aspects of the chemistry, mathematics, and computer science. For example, it is now common for chemistry programs to use basic linear algebra subroutines (BLAS), which are bundled in libraries that have been generally developed by mathematicians and computer scientists to perform well. By using these libraries, chemists can concentrate on the chemical/physical problem at hand instead of worrying about the performance of the BLAS.

The developers of NWChem [Kendall et al., 2000; Pacific Northwest National Laboratory, 2004b] and MPQC [Janssen et al., 1998; Sandia National Laboratories, 2004] have joined with others in the CCA effort to examine the use of components in quantum chemistry. The first challenge chosen was that of optimizing a molecular geometry using NWChem and MPQC to calculate the energy, gradient, and Hessian (function, first derivative, and second derivative, respectively), TAO [Benson et al., 2001; Benson et al., 2003b] for the optimization capability, and PETSc [Balay et al., 2003; Balay et al., 1997] or Global Arrays [Nieplocha et al., 1996; Pacific Northwest National Laboratory, 2004a] for the linear algebra and management of distributed data structures. The basic component diagram is shown in Figure 11.



Molecular geometry optimizations are a very common part of quantum chemistry applications, and each code has its own specializations to reach a structure in as few function evaluations as possible. This model makes it very difficult to reuse code between programs since the optimization routines can be very tied into the particular program. Our initial goal in this work was to decouple the optimization problem from the function evaluations so that multiple chemistry codes could be used with multiple optimization methods. Therefore, when a new optimization method is developed, it will be available to a wider chemistry community. This approach also allows users to have multiple chemistry programs available to them through a common interface, thereby allowing users to take advantage of the

different algorithms available for solving a particular chemical problem.

In order to accomplish this goal, of course, interfaces needed to be standardized. In this case, a decision was made to develop a general quantum chemistry interface that satisfied the NWChem and MPQC developers' needs as opposed to developing a broader interface with the whole chemistry community. This decision was made so that initial progress could be made and experience could be gained. In the future, we anticipate that other chemists will be involved, at which time the interface will need to be revisited. These interfaces use SIDL specifications and Babel to create server implementations for both NWChem, which uses Fortran, and MPQC, which uses C++. Other interfaces associated with the optimization model and with the linear algebra had already been created before the chemistry developers became involved. These interfaces were used with a few modifications as appropriate for the chemistry problem.

In this approach, a very coarse level of componentization was chosen to begin. Very high level capability, such as setting molecular geometries, wavefunction type, and basis sets and then computing energies, gradients, and Hessians is made available through the quantum chemistry interface. Now that the developers have experience with the CCA infrastructure, we will be developing lower level components, such as those for building Fock matrices, calculating properties, and other mathematical constructs such as eigensolvers and diagonalizers.

We are also currently exploring the use of the MCMD execution model for carrying out independent function evaluations simultaneously and for increasing flexibility in composing programs. This type of capability will be critical to applications that rely on methods such as Monte Carlo, combined quantum chemical methods (those that use varying levels of theory to model different parts of the chemical system), and dynamics simulations where multiple, relatively independent function evaluations are required.

14 The People Behind the CCA

The small grass-roots effort that launched the CCA has grown into a sizable community of computational science researchers developing and using CCA technology, represented by the CCA Forum. Members of this community have also successfully launched a variety of funded research projects that focus on various aspects of the development and use of the CCA.

The CCA Forum can be thought of as a combination of a standards body and a user group for the CCA. The Forum has met quarterly since 1998 and is open to all interested parties. (Meeting information is available on the Forum's

web site (<http://www.cca-forum.org> and on its primary mailing list [CCA Forum, 2004].) Meetings, as well as the mailing list, serve as an opportunity for discussion of issues and experiences relating to the development and use of the Common Component Architecture. The Forum also defines the CCA specification. Voting privileges are conferred on all who have attended two of the last three CCA Forum meetings as of the date of the proposal, and voting takes place via the web over a two week period. At the moment, the Forum has 24 voting members representing 11 different institutions.

The core group that launched the CCA Forum (Argonne, Lawrence Livermore, Los Alamos, Oak Ridge, Pacific Northwest, and Sandia National Laboratories, Indiana University, and the University of Utah) in 2001 created the Center for Component Technology for Terascale Simulation Software (CCTTSS) as part of the U. S. Department of Energy's (DOE) Scientific Discovery through Advanced Computing (SciDAC) program [U. S. Dept. of Energy, 2003]. The mission of the CCTTSS is to develop a better understanding of the most appropriate ways to use CBSE in scientific computing, as well as to develop the fledgling CCA ideas and prototypes into a production-quality environment suitable for use in large-scale applications. Thus, the CCTTSS acts as the anchor for CCA development.

However, the CCTTSS is far from the only funded project relating to the CCA. Other more focused efforts investigate performance monitoring and tuning and various aspects of distributed computing with sponsorship from the DOE, NASA, and the National Science Foundation. In almost all cases, CCA users have separate funding through grant proposals that may or may not have explicitly mentioned CCA.

15 Future Work

The Common Component Architecture is a long-term research and development effort with contributions from numerous separate projects, anchored by the Center for Component Technology for Terascale Simulation Software. It is impossible to capture the CCA-related research plans of all these projects and individuals; however, we can point to a number of enduring themes and highlight several specific initiatives that we expect to yield results in the near term.

15.1 Understanding of the Role of CBSE in HPC

One of the key motivations for pursuing the CCA in the first place was the fact that CBSE had made few inroads into HPC scientific computing with existing component environments. As a consequence, an important part of the

CCA effort is to gain a better understanding of how CBSE can be best used in this area. As part of this effort, CCA researchers work closely with component and application developers not only on the software itself, but also to gain information about the use of CBSE is effecting the process. The results of this work feed back into the design of the CCA and future component technologies, as well as contribute to the development of “best practices” and other guidance on how to use CBSE in scientific computing.

15.2 Development of the CCA Specification and Environment

As mentioned previously, the CCA specification is not a static document, but rather evolves as CCA researchers find that additions or changes are needed. Likewise, the tools that implement the CCA environment are also still very much under development. New tools to improve or extend the CCA environment are also under development by numerous researchers.

15.3 Development of Interfaces and Components

Another of the primary motivations for the development of the CCA is to increase software productivity by facilitating the reuse of components. In this context, a rich library of components that can be used to simplify the creation of new applications is an attractive feature. Section 11 is a good start, but it is also just a tiny fraction of what *could* be made available as reusable components. CCA researchers will continue to engage domain experts to design common interfaces to promote interoperability and to encourage the development of components implementing these interfaces.

15.4 Parallel Data Redistribution and Parallel Remote Method Invocation

Section 12 described in some detail our ongoing effort to develop generalized tools and capabilities for model coupling and related problems. Work on parallel data redistribution has already been used in several demonstrations and applications, and the research is progressing to incorporate the initial lessons learned into an improved interface design and software implementations. Work on parallel RMI is much newer, but we expect it to produce useful results in the near term.

15.5 Richer Interface Descriptions

At present, the CCA works with a very basic syntactic definition of component interfaces, which includes method names, argument types, and some information about memory management responsibilities. The definition of the semantics and behavior of the interface can currently be expressed only through comments in code. While this limitation is widespread within both component-based and traditional computing and is in no way unique to the CCA, it is clear that to fully realize the benefits of CBSE for complex scientific applications, richer interface descriptions that provide more complete, readily processable specifications of syntax, semantics, and behavior are highly desirable. The more complete the specification, the greater the confidence application developers can have that they are using components correctly [Baudry et al., 2001; Beugnard et al., 1999]. Expressing such interface specifications in a form amenable to machine processing allows automated verification of correct usage and operation. The *Design by Contract* approach [Meyer, 1997] to ensuring correctness relies on the use of assertions. The basic assertion mechanisms suitable for interface specifications are being integrated into Babel [Dahlgren et al., 2002; Dahlgren et al., 2003].

However, correct usage of a component goes beyond assertions about the arguments and return values of its methods. Another factor is whether the methods are being invoked in the correct order [Bronsard et al., 1997; Hamie, 1999]. This is a common issue for any complex process or computation. Real-world applications often require multiple steps in specific sequences. For example, a matrix must be initialized and assembled before computations on it can be performed. Unfortunately, it is easy for developers to forget or misplace a step, making debugging large or complex applications more difficult.

The challenges to richer descriptions include capabilities and limitations relating to expressiveness and extension within the context of interface specifications. For example, only those methods listed in the specification can be included in the assertions, even if underlying dependencies exist within the implementation. In addition, an extending class may need to expand on the sequencing constraints of its ancestors through the insertion of additional steps in the middle of an inherited step. Furthermore, the performance costs associated with retaining an adequate level of assertion checking in deployed components is a concern for both component developer and user. Questions that are currently being explored focus on a new approach to assertion checking to mitigate performance issues in deployed components. Long term research will focus on expressiveness and scalability.

15.6 Dynamic Code Insertion and Remote Method Invocation in Babel

The fact that Babel sits between the user and provider of a port puts it in a very useful position to allow it to perform a wide range of operations in conjunction with method calls. The verification of assertions described above is one example. But one can use Babel's unique position to dynamically interpose code between caller and callee in a more general fashion. Hooks are being added to Babel to allow this to be done under user control. A prototype version of this capability has been used to support integration of the Tuning and Analysis Utilities (TAU) [Malony and Shende, 2000; University of Oregon, 2003] into the CCA environment, thereby “automatically” instrumenting all inter-port method calls with calls to the TAU routines to start and stop gathering of performance information. Other applications of this capability are under investigation, including limited forms of aspect-oriented programming [Aspect-Oriented Software Association, 2004].

Similarly, the boundary between languages, and the boundary between components, is a logical place to implement remote method invocation (RMI) to provide more direct support for distributed computing. From Babel's perspective, RMI is just like any other language backend. Because Babel is between caller and callee, it can serialize the method invocation over a wire protocol and invoke the method in another process or on another machine.

SIDL already supports method arguments as `in`, `out` or `inout`, and these designations allow the in-process and RMI layer to optimize the method invocation. Arguments can also be designated as `copy` to indicate that Babel should serialize the object for a distributed call rather than only passing a remote reference to the object. Finally, methods can be designated as `oneway` so the caller can dispatch the call over the wire without waiting for a return code indicating completion of the method.

We are currently integrating an RMI backend using the Proteus multiprotocol message library [Chiu et al., 2002], which is implemented in C++. Once Proteus has been integrated into Babel, it will enable interoperability between the CCaffeine and XCAT frameworks.

15.7 Computational Quality of Service

The relative maturity of component-based software infrastructures encourages users to look beyond syntactically connecting components to using higher-level information about component properties to compose applications. Some of these properties pertain to the *quality* of the services used and provided by components. As more scientific components

are developed, many applications are presented with a pool of algorithms with similar functionality but different resource requirements, performance, accuracy, and stability. For example, at the heart of many PDE-based simulations, a set of nonlinear equations must be solved, usually requiring the solution of many linear systems of equations, whose characteristics may vary throughout the simulation. By formally specifying the *quality* requirements of the nonlinear solution method, as well as the capabilities of different linear solution methods, one can at least partially automate the complex task of assembling components into applications and adaptively switching component instances at runtime.

Some nascent efforts on defining a model for the QoS-based composition of high-performance numerical components address various aspects of a possible architecture design that enables automated application composition and run-time application adaptivity [Hovland et al., 2003; Norris et al., 2004]. The draft QoS architecture defines specifications and services for component characterization, component proxy services, component replacement, decision-making functionality, and archival and retrieval of execution information. The quality metrics in this architecture include traditional ones, such as reliability and computational cost, as well as metrics that are uniquely important to scientific computations, such as accuracy and rates of convergence. The long term goal of this research is to guide the design of a methodical and automated approach to component application composition and run-time adaptation.

16 Conclusions

Component-based software engineering is a natural approach to facilitate the development of complex, large-scale, high-performance scientific simulations. Its adoption in this area has been limited by the lack of a component model that meets the special needs of the HPC community; however, the Common Component Architecture is being developed specifically to address this issue. The CCA supports sequential, parallel, and distributed computing paradigms with minimal performance overhead and with a special emphasis on facilitating the incorporation of existing code.

Although the project is far from complete, the CCA specification and associated tools have reached a level of maturity required for adoption by a number of projects across a range of scientific disciplines. We have provided a brief overview of several such efforts in combustion, climate modeling, and quantum chemistry, as well as offered a glimpse at the wide range of components currently available or under development.

Our understanding of how best to utilize CBSE in scientific computing advances continuously through our interactions with the various projects utilizing the CCA. However, the field is broad and highly varied, and it will take some

time to gain the necessary breadth and depth of experience. There are also many opportunities on the research front of CBSE itself, and we outlined a number of areas that CCA researchers are exploring. Nevertheless, clearly much research and development remain to be done to bring the full benefits of CBSE to scientific computing and ultimately to make it part of the mainstream for computational science.

17 Acknowledgments

The CCA has been under development since 1998 by the CCA Forum and represents the contributions of many people, all of whom we gratefully acknowledge. We further acknowledge our collaborators outside the CCA Forum and the early adopters of the CCA for the important contributions they have made both to our understanding of CBSE in the high-performance scientific computing context and to making the CCA a practical and usable environment.

This work has been supported in part by the U. S. Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) initiative, through the Center for Component Technology for Terascale Simulation Software, of which Argonne, Lawrence Livermore, Los Alamos, Oak Ridge, Pacific Northwest, and Sandia National Laboratories, Indiana University, and the University of Utah are members. Members of the SciDAC center Computational Facility for Reacting Flow Research have also contributed to this paper.

Research at Argonne National Laboratory was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

Some of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Oak Ridge National Laboratory is managed by UT-Battelle, LLC for the US Dept. of Energy under contract DE-AC-05-00OR22725.

This research was performed in part using the Molecular Science Computing Facility (MSCF) in the William R. Wiley Environmental Laboratory at the Pacific Northwest National Laboratory (PNNL). The MSCF is funded by the Office of Biological and Environmental Research in the U.S. Department of Energy. PNNL is operated by Battelle for the U.S. Department of Energy under contract DE-AC06-76RLO 1830.

Research at Binghamton University is sponsored by grant DE-FG02-02ER25526 from the MICS program of the

U.S. Dept. of Energy, Office of Science.

Some of research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology and by Northrop Grumman/TASC with funding provided by NASA's Computation Technologies (CT) Project, part of the Earth Science Technology Office (ESTO), under a contract with the National Aeronautics and Space Administration.

Research at the University of Oregon is sponsored by contracts (DE-FG03-01ER25501 and DE-FG02-03ER25561) from the MICS program of the U.S. Dept. of Energy, Office of Science.

Research at the University of Utah is also sponsored by the National Science Foundation under contract ACI0113829.

References

- Allan, B., Armstrong, R., Lefantzi, S., Ray, J., Walsh, E., and Wolfe, P.: 2003, *Ccaffeine – a CCA Component Framework for Parallel Computing*, <http://www.cca-forum.org/ccafe/>
- Allan, B. A., Armstrong, R. C., Wolfe, A. P., Ray, J., Bernholdt, D. E., and Kohl, J. A.: 2002, *Concurrency and Computation: Practice and Experience* **14(5)**, 1
- Allen, G., Benger, W., Goodale, T., Hege, H., Lanfermann, G., Merzky, A., Radke, T., Seidel, E., and Shalf, J.: 2000, in *High Performance Distributed Computing (HPDC)*, pp 253–260, IEEE Computer Society
- Anonymous: 2003, *Jython*, <http://www.jython.org/>
- Anonymous: 2004, *AVS Kernel Overview*, http://www.agocg.ac.uk/reports/visual/vissyst/dogbo_28.htm
- Argonne National Laboratory: 2003, *MPICH homepage*, <http://www.mcs.anl.gov/mpi/mpich/>
- Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., and Smolinski, B.: 1998, in *Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*
- Aspect-Oriented Software Association: 2004, *Aspect-Oriented Software Development*, <http://www.aosd.net/>
- Balay, S., Buschelman, K., Gropp, W., Kaushik, D., Knepley, M., McInnes, L., Smith, B. F., and Zhang, H.: 2003, *PETSc Users Manual*, Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, <http://www.mcs.anl.gov/petsc>
- Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F.: 1997, in E. Arge, A. M. Bruaset, and H. P. Langtangen

- (eds.), *Modern Software Tools in Scientific Computing*, pp 163–202, Birkhauser Press, also available as Argonne preprint ANL/MCS-P634-0197
- Baudry, B., Traon, Y. L., and Jezequel, J.: 2001, in *Proceedings of the 7th International Software Metrics Symposium*, pp 272–284
- Beazley, D.: 2003, *SWIG homepage*, <http://www.swig.org/>
- Beckman, P., Fasel, P., Humphrey, W., and Mniszewski, S.: 1998, in *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computation*
- Benson, S., Krishnan, M., McInnes, L., Nieplocha, J., and Sarich, J.: 2003a, *Using the GA and TAO Toolkits for Solving Large-Scale Optimization Problems on Parallel Computers*, Technical Report ANL/MCS-P1084-0903, Argonne National Laboratory, submitted to ACM-TOMS, ftp://info.mcs.anl.gov/pub/tech_reports/reports/P1084.pdf
- Benson, S., McInnes, L. C., and Moré, J.: 2001, *ACM Transactions on Mathematical Software* **27**, 361
- Benson, S., McInnes, L. C., Moré, J., and Sarich, J.: 2003b, *TAO Users Manual*, Technical Report ANL/MCS-TM-242 - Revision 1.5, Argonne National Laboratory, <http://www.mcs.anl.gov/tao/>
- Bernholdt, D. E., Armstrong, R. C., and Allan, B. A.: 2004, in *Proc. of HPCA Workshop on Productivity and Performance in High-End Computing (P-PHEC 2004)*, Madrid, Spain, IEEE Computer Society
- Bernholdt, D. E., Elwasif, W. R., and Kohl, J. A.: 2002a, in D. Kranzlmüller, P. Kacsuk, J. Dongarra, and J. Volkert (eds.), *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 9th European PVM/MPI User's Group Meeting Linz, Austria, September/October 2002. Proceedings*, Vol. 2474 of *Lecture Notes in Computer Science*, pp 260–270, Springer
- Bernholdt, D. E., Elwasif, W. R., Kohl, J. A., and Epperly, T. G. W.: 2002b, in *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries (POHLL-02)*
- Bertrand, F. and Bramley, R.: 2004, in *Proceedings of HIPS 2004, 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, IEEE Press, Santa Fe, NM, to appear
- Beugnard, A., Jezequel, J., Plouzeau, N., and Watkins, D.: 1999, *IEEE Computer* **32(7)**, 38
- Birrell, A. D. and Nelson, B. J.: 1984, *ACM Transactions on Computer Science* **2(1)**, 39
- Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry,

- G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C.: 1997, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA
- Box, D.: 1997, *Essential COM*, Addison-Wesley Pub Co
- Bronsard, F., Bryan, D., Kozaczynski, W. V., Liongosari, E., Ning, J. Q., Olafsson, A., and Wetterstrand, J. W.: 1997, in *Proceedings of the 1997 Symposium on Software Reusability*, pp 19–29
- Brown, D. L., Henshaw, W. D., and Quinlan, D. J.: 1999, in *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pp 58–67, SIAM
- CCA Forum: 2003, *CCA Specification*, <http://cca-forum.org/specification/>
- CCA Forum: 2004, *cca-forum@cca-forum.org mailing list*, <http://www.cca-forum.org/mailman/listinfo/cca-forum/>
- Chiu, K., Govindaraju, M., and Gannon, D.: 2002, in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pp 1–9, IEEE Computer Society Press
- Chow, E., Cleary, A., and Falgout, R.: 1999, in *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, SIAM
- Christensen, E. et al.: March 2001, *Web Services Description Language (WSDL) 1.1*, <http://www.w3.org/TR/wsdl/>
- Colella, P.: 2003, *An Algorithmic and Software Framework for Applied Partial Differential Equations Center (APDEC)*, <http://davis.lbl.gov/APDEC/>
- Dahlgren, T., Epperly, T., Kohn, S., and Kumfert, G.: 2002, *Introducing Design-by-Contract to SIDL/Babel*, Technical Report UCRL-PRES-150102, Lawrence Livermore National Laboratory, <http://www.llnl.gov/CASC/components/docs.html>
- Dahlgren, T., Epperly, T., and Kumfert, G.: 2003, *Babel/SIDL Design-by-Contract: Status*, Technical Report UCRL-PRES-152674, Lawrence Livermore National Laboratory, <http://www.llnl.gov/CASC/components/docs.html>
- Dahlgren, T., Epperly, T., and Kumfert, G.: 2004, *Babel User's Guide*, CASC, Lawrence Livermore National Laboratory, version 0.9.0 edition
- Damevski, K.: 2003, *Master's thesis*, The University of Utah

- Denis, A., Pérez, C., and Priol, T.: 2001, in C. A. Lee (ed.), *Proceedings of the 2nd International Workshop on Grid Computing*, Vol. 2242 of *Lecture Notes in Computer Science*, pp 14–25, Springer-Verlag, Berlin
- Denis, A., Pérez, C., Priol, T., and Ribes, A.: 2003, *Distributed Systems Online* 4(2), http://dsonline.computer.org/0302/f/pri_print.htm
- Drummond, L. A., Demmel, J., Mechoso, C. R., Robinson, H., Sklower, K., and Spahr, J. A.: 2001, in *Proceedings of the International Conference on Computational Science*, pp 31–40
- Edjlali, G., Sussman, A., and Saltz, J.: 1997, in *International Parallel Processing Symposium*, IEEE Computer Society Press, Geneva, Switzerland
- Englander, R.: 1997, *Developing Java Beans*, O'Reilly and Associates
- Falgout, R. et al.: 2003, *hypre*, <http://www.llnl.gov/CASC/hypre/>
- Forum, M. P. I.: 1994, *International Journal of Supercomputer Applications and High Performance Computing* **8(3/4)**, 159
- Foster, I. and Kesselman, C.: 1998, *The GRID: Blueprint for a New Computing Infrastructure*, Morgan-Kaufmann
- Foster, I., Kesselman, C., Nick, J., and Tuecke, S.: 2002, *Computer* 35(6)
- Geist, G. A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V.: 1994, *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA
- Geist, G. A., Kohl, J. A., and Papadopoulos, P. M.: 1997, *The International Journal of High Performance Computing Applications* **11(3)**, 224
- Glimm, J., Brown, D., and Freitag, L.: 2001, *Terascale Simulation Tools and Technologies (TSTT) Center*, <http://www.tstt-scidac.org/>
- Global Grid Forum: 2003, *The Open Grid Services Infrastructure Working Group*, <http://www.gridforum.org/ogsi-wg/>
- Gosling, J., Joy, B., and Steele, G.: 1996, *The Java Language Specification*, Available at <http://java.sun.com/>
- Govindaraju, M., Krishnan, S., Chiu, K., Slominski, A., Gannon, D., and Bramley, R.: 2003, in *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*
- Grimshaw, A., Ferrari, A., Knabe, F., and Humphrey, M.: 1999, *IEEE Computer* 32(5)
- Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., and Nielsen, H. F.: 2003, *SOAP Version 1.2*, <http://www.w3.org/TR/2003/NOTE-SOAP-1.2-20030514/>

[//www.w3.org/TR/soap12-part1/](http://www.w3.org/TR/soap12-part1/)

Guilyardi, E., Budich, R. G., and Valcke, S.: 2002, in *Proceedings of Realizing TeraComputing - Tenth Workshop on the Use of High Performance Computing in Meteorology*

Hamie, A.: 1999, in *Proceedings of 6th Asia-Pacific Software Engineering Conference (APSEC '99)*, pp 376–383

Harper, L. and Kauffman, B.: 2004, *Community Climate System Model*, <http://www.cesm.ucar.edu/>

Henshaw, W. et al.: 2002, *Overture*, <http://www.llnl.gov/CASC/Overture/>

Heroux, M., Bartlett, R., Hoekstra, V. H. R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., and Williams, A.: 2003, *An Overview of Trilinos*, Technical Report SAND2003-2927, Sandia National Laboratories

Heroux, M. et al.: 2004, *Trilinos*, <http://software.sandia.gov/Trilinos>

Hindmarsh, A. and Serban, R.: 2002, *User Documentation for CVODES, An ODE Solver with Sensitivity Analysis Capabilities*, Technical Report UCRL-MA-148813, Lawrence Livermore National Laboratory, <http://www.llnl.gov/CASC/sundials/>

Hoare, M. R.: 1979, *Advances in Chemical Physics* **40**, 49

Hovland, P., Keahey, K., McInnes, L. C., Norris, B., Diachin, L. F., and Raghavan, P.: 2003, in *Proceedings of Workshop on QoS in Component-Based Software Engineering, Software Technologies Conference*, Toulouse, France

Janssen, C. L., Nielsen, I. M. B., and Colvin, M. E.: 1998, *Encyclopedia of Computational Chemistry*, Chapt. Parallel Processing for ab Initio Quantum Mechanical Methods, John Wiley & Sons, Chichester, UK

Johnson, C. and Parker, S.: 1995, in H. Meuer (ed.), *Supercomputer '95*, pp 2–19, Springer-Verlag

Katz, D. S., Tisdale, E. R., and Norton, C. D.: 2002, in *Recent Advances in Computational Science & Engineering: Proceedings of the International Conference on Scientific & Engineering Computation (IC-SEC) 2002*, pp 353–356, Imperial College Press

Keahey, K.: 1996, *Ph.D. thesis*, Indiana University - Bloomington

Keahey, K., Fasel, P., and Mniszewski, S.: 2001, in *Proceedings of the High Performance Distributed Computing Conference*, San Francisco, CA

Keahey, K. and Gannon, D.: 1997, in *Proceedings of the High Performance Distributed Computing Conference*, pp 31–39

- Kendall, R. A., Apra, E., Bernholdt, D. E., Bylaska, E. J., Dupuis, M., Fann, G. I., Harrison, R. J., Ju, J. L., Nichols, J. A., Nieplocha, J., Straatsma, T. P., Windus, T. L., and Wong, A. T.: 2000, *Computational Physics Communication* **128**, 260
- Keyes, D.: 2004, *Terascale Optimal PDE Simulations (TOPS) Center*, <http://tops-scidac.org/>
- Killeen, T., Marshall, J., and da Silva, A.: 2003, *Earth System Modeling Framework homepage*, <http://www.esmf.ucar.edu/>
- Kohl, J. A.: 1997, *ORNL Review, Special Issue on Advanced Computing* **30(3/4)**, 224
- Kohl, J. A. and Geist, G. A.: 1999, in *IASTED International Conference on Applied Modeling and Simulation*, Cairns, Queensland, Australia
- Kohl, J. A. and Papadopoulos, P. M.: 1995, in *High Performance Computing Symposium*, Montreal, CA
- Krishnan, S. and Gannon, D.: 2004, in *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, IEEE Press, Santa Fe, NM, to appear
- Kumfert, G.: 2003, *Understanding the CCA Specification using Decaf*, Technical Report UCRL-MA-145991, Lawrence Livermore National Laboratory, <http://www.llnl.gov/CASC/components/docs.html>
- Kumfert, G. and Epperly, T.: 2002, *Software in the DOE: The Hidden Overhead of "The Build"*, Technical Report UCRL-ID-147343, Lawrence Livermore National Laboratory
- Larson, J., Jacob, R., and Ong, E.: 2004a, *Model Coupling Toolkit*, <http://www.mcs.anl.gov/mct/>
- Larson, J. W., Jacob, R. L., Foster, I. T., and Guo, J.: 2001, in V. N. A. and J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. J. K. Tan (eds.), *Proceedings of the International Conference on Computational Science (ICCS) 2001*, Vol. 2073 of *Lecture Notes in Computer Science*, pp 185–194, Springer-Verlag, Berlin
- Larson, J. W., Norris, B., Ong, E. T., Bernholdt, D. E., Drake, J. B., Elwasif, W. R., Ham, M. W., Rasmussen, C. E., Kumfert, G., Katz, D. S., Zhou, S., DeLuca, C., and Collins, N. S.: 2004b, in *84th American Meteorological Society Annual Meeting*, American Meteorological Society, Seattle, Washington
- Larson, J. W., Rasmussen, C. E., Volberg, O., Ong, E. T., and Jacob, R. L.: 2004c, *From Large Coupled Model to a Component-based Application: Building Bridges between the Model Coupling Toolkit, the Earth System Modeling Framework, and the Common Component Architecture*, manuscript in preparation
- Lawrence Livermore National Laboratory: 2004, *Babel homepage*, <http://www.llnl.gov/CASC/>

`components/babel.html`

- Lee, J. and Sussman, A.: 2004, *Efficient Communication Between Parallel Programs with InterComm*, Technical Report CS-TR-4557 and UMIACS-TR-2004-04, University of Maryland, Department of Computer Science and UMIACS, A shortened version submitted to HPDC-13
- Lefantzi, S., Kennedy, C., Ray, J., and Najm, H.: 2003a, in *Proceedings of the Fall Meeting of the Western States Section of the The Combustion Institute*, Los Angeles, California, Distributed via CD-ROM
- Lefantzi, S. and Ray, J.: 2003, in *Proceedings of the Second MIT Conference on Computational Fluid and Solid Mechanics, June 17-20, 2003, Cambridge, MA*, Vol. 2, pp 1401–1405, Elsevier
- Lefantzi, S., Ray, J., and Najm, H. N.: 2003b, in *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France*, IEEE Computer Society, Distributed via CD-ROM
- Lewis, M., Ferrari, A., Humphrey, M., Karpovich, J., Morgan, M., Natrajan, A., Nguyen-Tuong, A., Wasson, G., and Grimshaw, A.: 2003, *Journal of Parallel and Distributed Computing* **63**, 525
- Liberty, J.: 2003, *Learning Visual Basic .NET*, O'Reilly and Associates
- Malony, A. D. and Shende, S.: 2000, *Distributed and Parallel Systems: From Concepts to Applications*, Chapt. Performance Technology for Complex Parallel and Distributed Systems, pp 37–46, Kluwer, Norwell, MA
- Meyer, B.: 1997, *Object-Oriented Software Construction*, Prentice Hall, Upper Saddle River, New Jersey 07458
- Microsoft Corporation: 1999, *Component Object Model Specification*, <http://www.microsoft.com/com/resources/comdocs.asp>
- Najm, H. N. et al.: 2003, *CFRFS homepage*, <http://cfrfs.ca.sandia.gov/>
- Nieplocha, J., Harrison, R. J., and Littlefield, R. J.: 1996, *J. Supercomputing* **10**(2), 169
- Norris, B., Balay, S., Benson, S., Freitag, L., Hovland, P., McInnes, L., and Smith, B.: 2002, *Parallel Computing* **28**(12), 1811
- Norris, B., Ray, J., Armstrong, R., McInnes, L. C., Bernholdt, D. E., Elwasif, W. R., Malony, A. D., and Shende, S.: 2004, *Computational Quality of Service for Scientific Components*, Submitted to International Symposium on Component-Based Software Engineering (CBSE7), Edinburgh, Scotland
- Object Management Group: 2002, *CORBA Component Model*, <http://www.omg.org/technology/>

documents/formal/components.htm

Ong, E., Larson, J., and Jacob, R.: 2002, in C. J. K. Tan, J. J. Dongarra, A. G. Hoekstra, and P. M. A. Sloot (eds.), *Proceedings of the 2002 International Conference on Computational Science*, Vol. 2330 of *Lecture Notes in Computer Science*, pp 748–757, Springer-Verlag, Berlin

OpenLDAP Foundation: 2003, *Lightweight Directory Access Protocol*, <http://www.openldap.org/>

Pacific Northwest National Laboratory: 2004a, *Global Array Toolkit homepage*, <http://www.emsl.pnl.gov:2080/docs/global/>

Pacific Northwest National Laboratory: 2004b, *NWChem homepage*, <http://www.emsl.pnl.gov/docs/nwchem/>

Parashar, M. et al.: 2004, *GrACE homepage*, <http://www.caip.rutgers.edu/~parashar/TASSL/Projects/GrACE/>

Parker, S. G. and Johnson, C. R.: 1995, in *Supercomputing '95*, IEEE Press

Parker, S. G., Weinstein, D. M., and Johnson, C. R.: 1997, in E. Arge, A. Bruaset, and H. Langtangen (eds.), *Modern Software Tools in Scientific Computing*, pp 1–44, Birkhauser Press

Ranganathan, M., Acharya, A., Edjlali, G., Sussman, A., and Saltz, J.: 1996, in *Proceedings of the 1996 International Conference on Supercomputing*, Philadelphia, PA

Rasmussen, C. E., Lindlan, K. A., Mohr, B., and Striegnitz, J.: 2001, in *2001 LACSI Symposium*

Rasmussen, C. E., Sottile, M. J., Shende, S. S., and Malony, A. D.: 2003, *Bridging the Language Gap in Scientific Computing: The CHASM Approach*, Technical Report LA-UR-03-3057, Advanced Computing Laboratory, Los Alamos National Laboratory

Ray, J., Kennedy, C., Lefantzi, S., and Najm, H.: 2003a, in *Proceedings of the Third Joint Meeting of the U.S. Sections of The Combustion Institute, March 16-19, 2003, Chicago, Illinois.*, Distributed via CD-ROM

Ray, J., Trebon, N., Shende, S., Armstrong, R. C., and Malony, A.: 2003b, *Performance Measurement and Modeling of Component Applications in a High Performance Computing Environment: A Case Study*, Technical Report SAND2003-8631, Sandia National Laboratory, Livermore, CA, Accepted, International Parallel and Distributed Computing Symposium, 2004, Santa Fe, NM

Reynders, J. et al.: 2001, *POOMA*, <http://www.codesourcery.com/pooma/pooma>

- Reynders, J. V. W., Cummings, J. C., Hinker, P. J., Tholburn, M., S. Banerjee, M. S., Karmesin, S., Atlas, S., Keahey, K., and Humphrey, W. F.: 1996, *POOMA: A FrameWork for Scientific Computing Applications on Parallel Architectures*, Chapt. 14, MIT Press
- Roman, E.: 1997, *Mastering Enterprise JavaBeans*, O'Reilly and Associates
- Sandia National Laboratories: 2004, *MPQC homepage*, <http://aros.ca.sandia.gov/~cljanss/mpqc/>
- Schmidt, D. C., Pyarali, I., and Harrison, T.: 1996, *USENIX Computing Systems* 9(4)
- Schmidt, D. C., Vinoski, S., and Wang, N.: 1999, *C++ Report* 11(9)
- Shende, S., Malony, A. D., Rasmussen, C., and Sottile, M.: 2003, in *Proceedings of International Workshop on Performance Modeling, Evaluation and Optimization, International Parallel and Distributed Processing Symposium*
- Slominski, A., Govindaraju, M., Gannon, D., and Bramley, R.: 2001, in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pp 1661–1667
- Smith, R. D., Dukowicz, J. K., and Malone, R. C.: 1992, *Physica D* 60(38)
- Sportisse, B.: 2000, *J. Comp. Phys.* **161**, 140
- Strang, G.: 1968, *SIAM J. Numer. Anal.* **5(3)**, 506
- Sun Microsystems: 2004a, *Enterprise JavaBeans Downloads and Specifications*, <http://java.sun.com/products/ejb/docs.html>
- Sun Microsystems: 2004b, *Java Remote Method Invocation (Java RMI)*, <http://java.sun.com/products/jdk/rmi/>
- Szyperski, C.: 1999, *Component Software: Beyond Object-Oriented Programming*, ACM Press, New York
- U. S. Dept. of Energy: 2003, *SciDAC Initiative homepage*, <http://www.osti.gov/scidac/>
- UDDI.org: 2003, *Universal Description Discovery and Integration of Business for the Web (UDDI)*, <http://www.uddi.org/specification.html>
- University Corporation for Atmospheric Research: 2004, *The Community Atmosphere Model (CAM) homepage*, <http://www.cesm.ucar.edu/models/atm-cam/>
- University of Oregon: 2003, *TAU: Tuning and Analysis Utilities*, <http://www.cs.uoregon.edu/research/paracomp/tau/>
- von Laszewski, G., Foster, I., Gawor, J., and Lane, P.: 2001, in *Concurrency and Computation: Practice and Experi-*

ence, Vol. 13(8-9), pp 643–662

Wehner, M. F., Mirin, A. A., Eltgroth, P. G., Dannevik, W. P., Mechoso, C. R., Farrara, J., and Spahr, J. A.: 1995, *J. Parallel Comp.* **21**, 1655

Wilde, T., Kohl, J. A., and Flanery, Jr., R. E.: 2002, in *International Conference on Computational Science (2)*, pp 864–873, Amsterdam, The Netherlands

Wilde, T., Kohl, J. A., and Flanery, Jr., R. E.: 2003, *Future Gener. Comput. Syst.* **19(5)**, 701

Zhang, K., Damevski, K., Venkatachalapathy, V., and Parker, S.: 2004, in *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, IEEE Press, Santa Fe, NM, to appear

Zhou, S.: 2003, *Coupling Earth System Models: An ESMF-CCA Prototype*, http://webserv.gsfc.nasa.gov/ESS/esmf_tasc/

Zhou, S., da Silva, A., Womack, B., and Higgins, G.: 2003, in *NASA Earth Science Technology Conference 2003*, College Park, MD, [http://esto.nasa.gov/conferences/estc2003/papers/A4P3\(Zhou\).pdf](http://esto.nasa.gov/conferences/estc2003/papers/A4P3(Zhou).pdf)